A MEASURE OF TEST CASE COMPLETENESS

BY

PATRICK JOSEPH WALSH

B.S., Mathematics, The Pennsylvania State University, 1968
M.A., Mathematics, The Pennsylvania State University, 1969
M.S., Systems & Information Science, Syracuse University, 1976

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy from the
Watson School of Engineering, Applied Science,
and Technology in the Graduate School of
the State University of New York
at Binghamton
1985

Accepted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy from the
T. J. Watson School of Engineering, Applied
Science, and Technology in the Graduate
School of the State University of
of New York at Binghamton

J. V. Cornacchio [illegible] April 20, 1985
Department of Computer Science
T. J. Watson School of Engineering,
Applied Science, and Technology

J. Diaz-Herrera [illegible] April 20, 1985
Department of Computer Science
T. J. Watson School of Engineering,
Applied Science, and Technology

L. Larson [illegible] April 20, 1985
Department of Computer Science
T. J. Watson School of Engineering,
Applied Science, and Technology

ACKNOWLEDGEMENTS

The author would like to thank his advisor, Dr. Joseph Cornacchio, for his guidance, encouragement, and suggestions for improving the readability of this dissertation; and to Dr. Lawrence Larson and Dr. Jorge Diaz-Herrera for their thorough reading of this manuscript. The author also wishes to thank his wife Cynthia and daughters, Meghan and Shannon for their patience during this endeavor. Finally the author wishes to thank his parents for placing emphasis on education.

ABSTRACT

When testing a computer program, it is not usually clear when to stop testing and, at the same time, have some level of assurance that the program is correct. This decision, as well as the selection of test cases, is often done in an ad hoc manner.

This dissertation addresses a technique that can be used to gain assurance that the quality of test cases are improving. First, a metric is developed to measure the effectiveness of a set of test cases developed using a particular testing approach, such as statement coverage, branch coverage, multiple condition coverage, path testing, cause-effect graphing and mutation analysis. A single measurement approach for test cases is developed, regardless of the test approach used to generate the test cases. This metric is used to evaluate both structural and functional methods for generating test cases. Next, a composite metric is constructed based on metrics developed for the approaches that were evaluated. This composite metric is shown, for the examples studied, to increase as the number of errors discovered increases.

That is, by adding more test cases in a manner that increases the composite metric, one is likely to find more program errors than by adding test cases in a random manner. In addition, by applying regression analysis, some of the components of the composite metric are shown to be a predictor of the reliability of the programs in the sample studied.

It is also shown that the cause-effect graphing and equivalent normal form approaches to test case generation produce the same test cases. The equivalent normal form method is more algorithmic and easier to implement.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1, INTRODUCTION

1.1 INTRODUCTION

A generalized metric is not available to measure the completeness of collections of test cases. As will be discussed later, individual methods of measurement are available for specific testing approaches; however, no overall measurement exists. The purpose of this dissertation is to develop a metric that can be used to assess the relative ability of test cases to uncover errors. The metric developed, based on an expanded completeness measure, can be used to evaluate any set of test cases, regardless of the method used to generate the test cases.

When testing a computer program, it is not usually clear when to stop testing and, at the same time, have some level of assurance that the program is correct. This decision, as well as the selection of test cases, is often done in an ad hoc manner. This paper will help toward developing an answer to these questions.

Chapter 1 is an introduction to testing that defines the terms to be used and explains why testing is necessary. In Chapter 2 many of the most common testing approaches are classified and described. This chapter is used as a base for Chapter 3, in which the metric for measuring test sets is developed. In Chapter 4, the concepts developed in Chapter 3 are implemented on four programs taken from the literature. It is shown that for all examples in this chapter, the higher the value of the derived metric, CA, the greater the number of known errors that are discovered. Chapter 5 verifies the results of the previous chapter by evaluating programs that were written for this purpose, instead of being taken from the literature. Also included in this chapter is the development of a statistical model to help convince us that the measurements taken are an estimate of the percent of errors that will be found. In Chapter 6, a summary of the results is given and ideas for future research are discussed.

Appendix A is simply a summary of the empirical data discussed in Chapter 4. The specifications given to the subjects who wrote the programs discussed in Chapter 5 are included in Appendix B. Appendix C shows the equivalence of the cause-effect approach and the equivalent normal form (ENF) approach for generating test cases. This is significant due to the availability of algorithms to implement the ENF methodology. Included in Appendix D are the details of the statistical analysis discussed in Chapter 5. Appendix E contains a Nassi-Shneiderman chart for the text reformatter program discussed in Chapter 4.

1.2 SUMMARY OF FINDINGS

A summary of the major findings of this dissertation follow:

i. The Cause-effect graphing approach is algorithmically equivalent to the ENF procedure used in the testing of circuits. The ENF approach is more algorithmic and easier to use for large applications. This is discussed briefly in section "2.2.4 Equivalent normal form" on page 34 and explained by means of an illustrative example in "Appendix C, Equivalence ENF and C-E procedures" on page 131.

ii. All test methods can be measured for completeness in a uniform manner with a range from 0 to 1. This is discussed in section " 3.2 An extension of the completeness measure" on page 64.

iii. The mutation concept is a super set of all other testing approaches that result in the building of test cases. This point is covered in section " 3.3 Test Methods as a Subset of Mutation" on page 70.

iv. There is a composite completeness measure, based on various testing approaches, that is monotone nondecreasing with the number of errors discovered. This is shown throughout "Chapter 4, Examples" on page 74 and summarized in section "4.5 Discussion of examples" on page 101.

v. The components of the composite completeness metric can be used to predict the reliability of a program. This is discussed throughout Chapter 5 and summarized in " 5.5 Results Obtained from the Model" on page 115.

Other contributions of lesser significance include the classification of testing methods shown in Figure 1 on page 18. A clear and simple classification scheme had not been previously developed.

1.3 INTRODUCTION TO TESTING

A survey of the literature (Myers, 1976; Miller, 1978; Glass, 1979; Wilson, 1983) reveals that between 40% and 75% of a program's life cycle cost is attributed to testing, retesting and error analysis activities. It is worth noting that only 5% of the software literature is devoted to testing (Gillion, 1983), a sign that this aspect of Computer Science is not as well understood as other areas. A justification for further research into the area can be seen by realizing that a 10% savings in test time would result in a $500 million saving per year (based on 500,000 programmers, a 40% average life cycle effort for testing and a $25,000 average salary). This does not include any saving based on using less computer time and the cost savings by having software available earlier.

The problems of unreliable software dates back to the first complex hardware and software installation, the SAGE System of the early fifties. The Air Force spent approximately ten billion dollars on the SAGE system but could only obtain a mean time to failure rate of about two hours. Almost all their failures (94%) were caused by software problems, 1% by hardware problems and 5% by operator error. In France seventy two meteorological balloons were incorrectly blown up due to a software error (Myers, 1976) and in 1962 the first space probe to Venus was aborted due to an error in one line of code in the on board computer (Manna and Waldinger, 1978). Recently the first manned space shuttle was delayed due to a software malfunction. These are examples of some of the problems that could have been avoided if the software had been properly tested. There is a list compiled of about twenty five serious problems caused by errors in software (Neumann, 1985), including three recent plane crashes.

There are several approaches used to increase the reliability of software. The first involves what is generally referred to as testing (an exact definition will be given later). The second is formal program proving techniques. A third approach to increase the reliability of software is a collection of design, analysis and management techniques that are applied, for the most part, prior to the end of the coding phase. Testing and program proving, on the other hand, are applied most often after a set of operational code is produced. There is a historical differentiation between design and code; however, some recent innovations, such as a detailed design using decision tables, that can be compiled into operational code can cause confusion on which phase one is in and; therefore, which tool to use. For the purposes of this dissertation, operational code is that document which a programmer will normally use to solve operational problems. For example, if one writes a decision table that is compiled into assembler language, and the assembler language is normally used to solve problems, then the assembler code would be considered the operational code; if the decision table document is normally used to solve problems, then it would be considered the operational code. Although this is not a precise definition, it is developed enough for the concepts to be clear.

It is clear that a large percentage of errors are introduced in the design phase of a project, 80% is one estimate (Alberts, 1978), so it is not surprising that a number of design methods influence the reliability of software. Top down development (Alberts, 1978); considering testing in the design phase by making provisions for software monitors (Hansen, 1978); restricting programming language facilities to increase the possibility of later showing that programs perform, as specified (Reynolds, 1980); generating a set of programming standards; decision tables (Hogger, 1977) are several of the methods that are likely to increase the reliability of the end product. For the most part their influence, although intuitively justified is intangible and has not been measured. During the coding phase factors that influence the end reliability are the use of higher level languages (Boehm, 1978) structured programming, design and code inspections (Fagan, 1978), and Nassi Shneiderman charts (Chapin, 1974). A comprehensive list (Glass, 1979; Myers, 1976) explains the methodologies that can be used to increase reliability. They include modular programming, change reviews, peer reviews, HIPO documentation, preventive maintenance and many other concepts.

Methods that increase the reliability of software, such as, design walk-throughs, code inspections, and structured programming, are often classified as static testing or validation techniques. In addition, there are a number of software tools (Boehm, McClean and Urfrig, 1978; Osterwiel, 1976; Ramamoorthy, and Ho, 1978; Howden, 1978) that statically analyze programs and report the suspected errors. These potential errors typically include output unit violations, subroutine calls with incorrect parameters, and variables that are not referenced. From one aspect, static error detection using automated tools is an extension of the compiler functions.

One of the goals of discussing software testing and program proving is to make the practicing programmer aware of these tools and concepts. Today testing is done mostly in an ad hoc manner based on the programmer's experience and intuition pertaining to the portions of the program that should be exercised. Too often testing is completed when the programmer has a good feeling about his code or, even worse, when the time allocated for testing is over.

1.3.1 Definition of terms

One of the problems in studying the software reliability and testing literature is the sometimes inconsistent definition and use of terms. It is common to treat software reliability and software testing in an informal manner without explicitly explaining the meaning of each. The following definitions are attempt to standardize the terms that are used in this paper:

Software testing, also referred to as testing, is the process of collecting and interpreting evidence about a program's suitability for operational use (Goodenough, 1980), not, as is commonly assumed, a process of finding and removing errors from a program.

Debugging is the process of locating and correcting a known error in a program (Myers, 1976). It is related to testing, since during the testing process, errors are discovered and must be corrected; however, no debugging tools or techniques are covered in this paper.

Program verification is the idea that one can state the intended effect of a program in a precise way that is not another program, and then prove rigorously that the program does (or does not) conform to this specification (Deutsch, 1973).

Software reliability is the probability that a software fault which causes a deviation from the required output by more than specified tolerances, in a specified environment, does not occur during a specified exposure period (Ramamoorthy and Bastani, 1982). It should be pointed out, that although testing is not a poor method of increasing software reliability, design tools and programming standards, also have a major effect on software reliability. Also note that the age of the software is not specifically mentioned. A program that satisfies this criteria is said tc be reliable.

Correctness or program correctness is satisfaction that a program's output meets specifications, independent of its use of computing resources, when operating under permitted conditions (Goodenough, 1980). The basis of the theory of testing (Goodenough and Gerhart, 1975) include several important definitions:

A program P is said to be completely correct with respect to f, the intended function, if and only if P computes only the correct values of f from arguments of f and is undefined for arguments outside the domain of f. This definition (Shankar, 1982) is a little more formal than Goodenough's correctness definition; however, a program is completely correct (Shankar's definition) if and only if it is correct (Goodenough's definition). Shankar also defines sufficient correctness if P computes values for arguments not belonging to the domain of f. It is important to note that a program may be correct and not reliable (for example an input parameter outside of permitted condition may result in a program fault) or reliable and not correct (for example the particular input parameter that will cause a fault is never run).

Robustness is the property of continuing to do something reasonable in the presence of unforeseen environmental changes (Chudleigh, 1982).

A set of inputs T is an ideal test for program P, relative to specification F, if the correct performance of P on T implies the program is correct on its entire input space.

A set of inputs T is said to be a reliable test, if and only if, its data selection criteria, C, ensures that every test satisfying C succeeds or every test fails. Reliability is really a measure of the data selection criterion. Tests of correct programs are 100% reliable.

A test data criteria C, and tests T are said to be complete, (T,C) if the data selection criteria, C, is used in selecting a particular set of test data. A more theoretical definition of complete (Howden, 1982) is as follows: P is a program, F is a set of functions associated with P. M is a mapping such that for each subset T of the domain P and each function f in F, M defines a subset of the domain f. Assume for each function f in F there is an associated set of functions S(f). Let T be the set of tests for P. Then T is a <u>complete set</u> of tests for P relative to F and [S(f): f in F].

The data selection criterion, C, is said to be <u>valid</u> if and only if, for every error in the program there exists a complete set of test data capable of revealing the error. This means that for each error in a program it is possible to select data that will uncover it, no guarantee is given that that data will be selected.

A test is <u>successful,</u> that is the test instance succeeds, if it produces normal program output when it is run. That is for each test case if the expected output is equal to the actual output then the test is successful, otherwise, an error has been discovered.

It is then our goal to select a data selection criterion, C, that is both reliable and valid. The primary theoretical point of Goodenough's and Gerhart's paper is called the fundamental theorem of testing and can be stated as follows: If there exists a consistent, reliable, valid and complete criterion for test set selection, for a program P and if a test set satisfying the criterion is such that all test instances succeed, then the program is correct (Adrion, Branstrad and Cherniavsky 1982).

1.3.2 Why is testing necessary

A question that should be asked is why not build programs correctly in the first place; that is, so they conform to the requirements and no rework is necessary. Unfortunately, present state-of-the-art techniques do not support that goal; although it is a worthwhile objective. Since, as stated before, a large amount of time is devoted to testing, it makes sense to develop some formal testing methodologies. The most straightforward method of insuring reliable software is to simply exercise the program by generating all possible inputs, running these inputs through the program, and comparing the computed results with the expected results. There are several problems with this approach, first, an oracle is required to determine the correct expected result. This is a practical problem that should not be overlooked; however, for the remainder of this paper the existence of such an oracle is assumed. The second problem is that, except for trivial programs, this method, referred to as exhaustive testing, is too time consuming to even consider. For example a simple 10 element sort, with the domain limited to the digits 0 to 9 would take 100 years to complete an exhaustive test assuming each test takes one second to execute. It would clearly be unacceptable to wait 100 years to guarantee 100% reliability. Test cases must be a subset of the exhaustive set picked in an intelligent manner to convince us that the software under test is reliable or the converse.

There are two classifications of program testing static and dynamic. In static testing (or more accurately static analysis) program execution is not required, while with dynamic testing, the program or part of it, is executed and the output is either automatically or manually compared with the anticipated results. Dynamic testing methods can be classified as those based on coverage criteria and those based on other measures. Coverage can be based on characteristics of the internal program structure (white or glass box testing) or those based on the functions provided (black box testing). Black box testing and white box testing methods can be combined in what is called grey box testing. Figure 1 on page 18 shows these classifications. This classification is slightly artificial, for example, weak mutation testing, a subset of mutation testing could also be considered a superset of branch testing. Also, symbolic execution could be considered a type of program proving. It should also be noted that the dynamic methods not based on coverage involve two steps. First the assertions are inserted or the mutant programs are developed, and second the test cases are developed, possibly using a coverage based model.

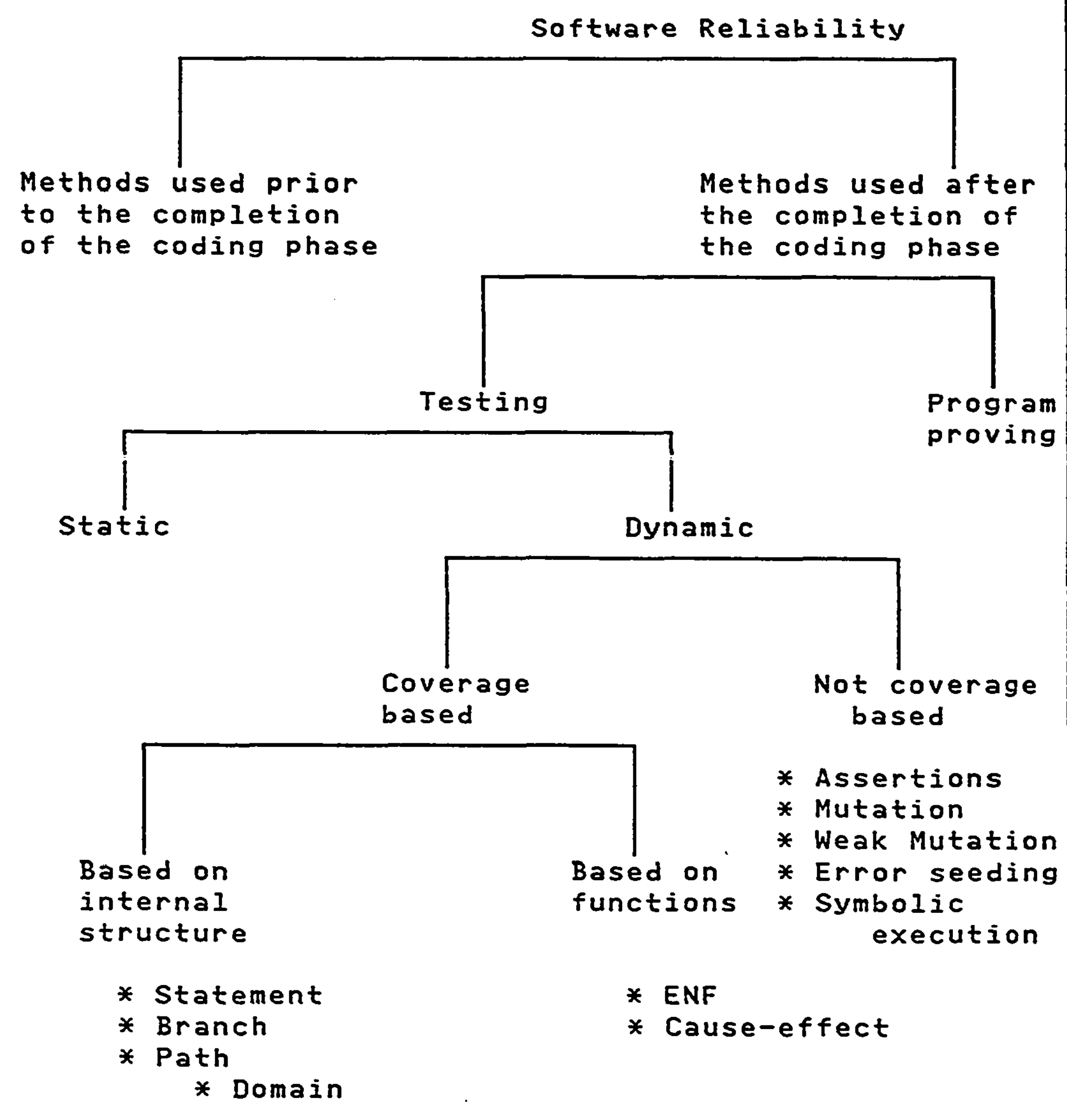


Figure 1. Classification of Reliability approaches

CHAPTER 2, CLASSIFICATION OF TESTING METHODS

2.1 STATIC TESTING

A number of errors, or possible errors in program construction can be discovered by having another program analyze the subject program. The basic approach to static analysis is to define these constraints and then write the programs that report violations. A common analysis involves the detection of variables that are referenced before they are initialized, or variables that are set and then never again used. Although, in the strictest sense these are not considered errors, their presence is an indicator of a potential problem. Due to the possibility of different paths leading to a single reference type instruction, the check to see if the variable has been initialized is not trivial (it may be initialized on several different paths). Algorithms have been developed to solve this problem (Fosdick and Osterweil, 1976), referred to as the live variable problem. Static analysis of this kind is based on first analyzing the subject program to generate a data flow graph. This graph (it can be assumed that a graph is generated for each variable) shows all the statements where the variable is initialized or referenced. Based on the program's control structure it can then be determined if any rules may be violated. There may be a path leading to a statement that references a variable that has not been initialized on the path. This may or may not be an error; perhaps the path is not logically possible. These, therefore, have to be flagged as possible errors using pure static analysis techniques, since the instructions are not executed so no determination can be made as to the logical possibility of executing that path. Other conditions may also be include in a static checker. For example division by a constant of zero; loops with no imbedded change in the loop variable, that is a type of infinite loop; and checking for coding standards.

Static analysis is actually an extension of the compiler's error checking capability, for example most compilers will flag any use of a variable that is not defined. Static analysis techniques can flag the possible use of a variable that is not initialized. Since the cost of running a static analyzer is relatively cheap, more of these functions will be built into future compilers.

In summary, static testing has the potential to detect a wide variety of errors that do not involve computational algorithms; the tools are easy to use and are relatively inexpensive to run. Another advantage of the static approach is the source of the error is usually also found. Many systems have been written to perform static checking, mostly for Fortran programs; sixteen operational tools are listed (Hiedler, et al., 1982). The DAVE system (Osterweil, 1978; Fosdick and Osterwiel, 1976) and the SQLAB system (Phoha, 1981) both incorporate static checking as part of their test tools. There have been articles published (Howden, 1978) that cover an introduction to static analysis.

2.2 DYNAMIC TESTING

Dynamic testing requires that the program or part of it be executed in one form or another. This is usually accomplished by executing the program via test cases, which are selected based on criteria for the approach being used. All of these approaches lead to the development of a set of test cases that, based on their individual criteria, provide some assurance that the program is being properly exercised and is therefore more reliable. Several comparisons of their relative usefulness have been published and will be discussed later; however no overall metric to measure the reliability gained by a given set of test cases has been developed. The statement coverage criteria, branch coverage, multiple condition coverage and the path criteria are covered. Also included are the dynamic methods of cause - effect graphing, mutation analysis, and error seeding.

2.2.1 Statement coverage

One method of selecting a subset of the exhaustive set of possible inputs is to select them in a manner that will assure that all instructions are executed at least once. This is referred to as statement coverage. Although, this is a rather weak condition, it is certainly required, for unless there are instructions in the program that can never be logically executed (a problem in itself), one can have no confidence that the instructions that are not executed are correct. A slightly stronger selection requirement called branch coverage requires that in addition to all statements being executed at least once, every branch direction must be traversed at least once. The branch coverage condition is stronger than statement coverage because additional test cases will be required to exercise the branch directions that do not result in any unique instructions being executed. For example a false branch that simply branches around the first instruction along the true path.

In order to determine if a given set of test cases satisfy the statement or branch coverage criteria software or hardware probes must be inserted into the system. The first step in inserting software probes is to generate a control graph (decision to decision path) of the program, and then inserting counters after each leg of all branch instructions. After the test cases are executed the counters should be displayed and all zero value counters used to develop additional test cases. There is not an algorithm that can be generally used to determine the input required to exercise a specific branch. This problem has been solved for some specific cases (Huang, 1978). More efficient, in some cases, algorithms have been developed to minimize the number of software probes required (Probert, 1982). This reduces one of the concerns with software probes, the added execution time spent updating them and the possibility of that delay hiding a timing error. This is especially a potential problem when working with real time programs. The PROBE system (Probert, 1982) is an example of a implementation of these ideas. An article on how to insert software probes (Wang, 1981) has been written.

Hardware probes can be developed along the same lines. They are more expensive; however, the problems encountered by changing the program's timing are avoided. Branch coverage is sometimes used as a trivial path coverage when it is too difficult to develop the stronger path cover. An example of this is a system developed to test some Motorola 6800 microprocessor programs (Yaccob and Hartley, 1981).

Statement coverage has been defined (Miller, 1977) as C1 coverage, and branch coverage as C2 coverage, C0 coverage is that based entirely on the programmer's intuition. An additional criterion called multiple condition coverage (Myers, 1976) requires branch coverage plus enough test cases to cover all possible combinations of condition outcomes in any direction. Multiple condition coverage is stronger than branch coverage when compound decisions are found in the program.

2.2.2 Path testing

Path Testing requires that each path through the program be exercised by at least one test case. There are three major problems with this requirement, first some paths may be logically impossible to generate a test case to execute, second, some paths may be generated that are never possible to execute in practice (these are called infeasible) third, generally the number of possible paths is very large. This is due to the fact that unique paths are generated for loop iteration, that is traversing a particular loop 100 times is a different path then traversing it 101 times. In the path testing approach the paths are usually divided into a finite and manageable number of classes with at least one test case generated from each class. In an unabridged form the path testing criteria is at least as strong as the branch coverage criteria. Path testing, even with the unrealistic assumption that every path in a program is tested does not detect all errors (Howden, 1978). It is shown that, for some published programs, 100% path testing does uncover a majority of the errors.

Some systems have been written that address the generation of test cases to exercise individual paths. The DAVE system (Clark, 1976) will attempt to generate data for a given path; however, the path (decision to decision graph) must be available and passed to the system. In order to determine the data to exercise a path its input constraints must be linear (since, in general it is impossible to find the input that causes a particular instruction, and therefore path to be executed). This system has the unique feature that symbolic execution is used to execute the path. The concept of symbolic execution is explained in "2.2.8 Symbolic execution" on page 46. The resulting set of equations is then solved, if possible, to determine the required input. Variable references are not allowed since they are difficult to symbolically execute.

The CASEGEN system (Ramamoorthy, Ho and Chen, 1979) operates very similar to the DAVE system, the paths are generated by some other means and then symbolically evaluated to find the proper inputs. A method of array reference is proposed to postpone the symbolic evaluation of a variable array reference until test data is generated. The RSVP system (Miller, Paige and Benson, 1978) is implemented using a tree representing the iteration of a program, the decision to decision paths are then printed as an aid to the person doing the testing.

A backtracking technique (Miller and Melton, 1978), instead of symbolic execution, is sometimes used to generate a test case for each path once the directed graph of the program flow is generated. A disadvantage of this method is that manual intervention may be necessary when iteration is involved (which is the case with most programs).

It has been shown (Gabow, Maheshwari and Osterweil, 1976) that there is an efficient way, based on graph theory, to find a path from one statement to another, through a specific set of vertices or to show that no such path exists. One of the problems listed earlier was that some paths through the program, although semantically possible can never be logically executed. These paths if generated are useless and, in fact, cause effort to be used to try to satisfy their constraints, an impossible task.

One approach to this problem is to generate sets of mutually unexecutable pairs of branches in a program (for example number > 1 and number x number < number). These pairs are referred to as impossible pairs and currently have to be manually generated after analysis of the programs structure. Gabow has shown that the IPP (impossible pair constrained path) is NP complete. This means that an analysis that computes paths and insures that no impossible pairs are included in any path is exponential in run time.

The primary problem when developing a path testing strategy is is to determine the criteria for path selection from the possibly infinite set of paths. A common approach is to limit those paths involving loops to: zero, one and the maximum number of iterations. It has been statistically shown, (Duran and Wiorkowski, 1980) a counter intuitive result, that insuring the testing of all paths does not give a better assurance of program correctness. A concise summary of the path testing problem is that it is enormously difficult and therefore can only be treated with proper reserve (Yacco, 1981).

The basis for statement, branch and path testing is a representation of the program structure called a directed graph (digraph) or decision to decision paths. Based on graph theory it is essentially a flowchart with only decision portions of the program included. It is a relatively simple procedure, given a program and the semantic rules to generate a digraph. Cyclomatic trees (Stickney, 1978), an improved version of a digraph can be used to generate more efficient test cases and to ease the placement of probes. Although it is possible to generate a digraph for each program it is not possible to write a general algorithm to generate the input that will cause a specific instruction to be executed. This is of particular concern in the methods that are based on coverage.

It has been shown (Tia, 1980) that statement, branch and path testing are not sufficient to demonstrate a program is correct. This is proven by showing the time complexity for some simple programming constraints is much higher than the time complexity of the above approaches. Two new criteria are developed based on the domain strategy, Cpath to test paths and Cprog to test programs. Cpath basically requires the selection of test points from each section of the partition of the input space as well as the boundary conditions. Cprog is an extension of this concept. These two criteria are not meant to guarantee the production of correct programs; however, their purpose is to serve as a guide for test case selection.

A special type of path testing called domain testing has been proposed (White and Cohen, 1980). Each path has a domain, or set of program inputs that cause that path to be executed. The domain testing concept is to select test values that are near the boundary between different paths. There are two types of possible program errors involving the selection of paths. The first, selection of the incorrect path is addressed by domain testing, the second, the missing path cannot be discovered using this approach. The underlying concept is that points near the boundary of a path domain are more likely to generate errors. Since this approach has the same drawbacks as general path testing, for example the handling of loops, it should not be used alone, but in conjunction with other methods. Several measures of how serious a domain error is were given by White and later expanded (Clarke, Hassell and Richardson, 1982).

2.2.3 Cause-effect graphing

Another method used to generate a complete set of test cases is cause-effect graphing (Elmendorf, 1976). All the causes (input conditions or system transformations) are identified and given a unique identifying number. Next, all effects (output conditions or changes in the system state) are identified and numbered. A graph is then generated by linking the causes to the effects with the proper logical relationships. The relationships used are AND, OR, identity and NOT as well as various constraint symbols. Parentheses are used to indicate the scope of the AND's and OR's.

As the size and ability to work with the cause-effect graphs increases quickly as the program specifications becomes more complex, the resulting cause-effect graph, developed from the Nassi-Shneiderman charts become very complex and difficult to generate. The next step in using the cause-effect graphing methodology is to generate a limited entry decision table that represents the portions of the graph that makes each output condition true (one at a time). The method used is to sequentially select each effect to be true and to trace back through the graph to find all combinations of causes that will make the given effect true. Each combination of effects is recorded in the decision table as a row. Some possible combinations may be ignored as they are not all necessary to generate the test cases and there may be an unreasonable number of combinations. All possible combinations may in fact mask certain causes. As an example, if four conditions are ORed together, it is only necessary to iterate four input conditions to make this true (each input true, while the others are false) instead of the fifteen possible combinations that make the output true. The final step is to convert the columns of the decision table into test cases. This is accomplished in a trial-and-error manner by inspecting the decision table and generating a test case for each column.

For example, if input A and input B combine to form condition C; and if condition C or Input D cause output E, then the cause-effect diagram would be as shown in Figure 2 on page 35. This would yield the limited entry decision table shown in Figure 3 on page 36. Next the four rows of this table are used as attributes for the test cases to be developed.

Cause-effect graphing is used as a procedural method to generate test cases. In addition, insight is gained into the problem to be solved by converting the specifications into the Boolean graph. It can also assist in the discovery of incomplete and inconsistent specifications. At least three tools are available today to automate the cause-effect graph process: TELDAP, developed by IBM; CEGAR, developed by The Bank of America, and one developed by Hitachi.

2.2.4 Equivalent normal form

Although hardware and software test generation concepts and functions are completely different, certain hardware concepts can be applied to software. One advantage of software over hardware is that software

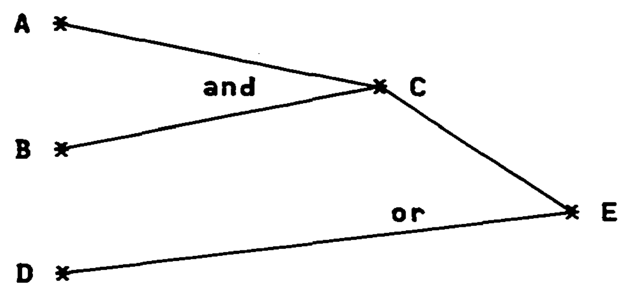


Figure 2. Example of Cause-effect graph

cannot develop any defects. The next method to generate a set of test cases is based on the equivalent normal form (ENF) of a hardware circuit (Friedman and Menon, 1979). We can apply the hardware ENF procedure to software by representing the program as a collection of hardware gates as was done in the in the cause-effect approach. The ENF is developed by expressing the output of each gate (output conditions or intermediate states for software) as a function of the inputs and at the same time preserving the identity of each gate. For example, if the inputs to an AND gate are A and B and the output is C, then the ENF for C would be represented as follows: ENF(C)= (A AND B)[C].

A	0	-	1	-
B	-	0	1	-
D	0	0	-	1
E	0	0	1	1

Figure 3. Decision Table for Cause-effect approach

If C and D are then inputs to an OR gate with output E, then the ENF for E is as follows:

ENF(E)= (C OR D)[E] = ((A AND B)[C] OR D)[E].

Figure 3 is a graphical representation of this relationship. Each character (for example, A[C]) is called a term. Terms connected by ANDS are called literals. The next step is to test each literal in an ENF for stuck at 0 by assigning 1's to all literals in the term containing it and making all other terms equal to zero. It is only necessary to test one literal per term using this method (testing more will result in duplicate tests). The ENF algorithm also requires that tests be generated for the stuck at 1 fault (a false output is expected) for all output conditions.

By selecting the inputs and outputs, as in the cause-effect graphing approach, one goes through the same mechanical procedures as in cause-effect graphing; the test case criteria generated is also identical. The equivalent normal form approach is more algorithmic and is, therefore, simpler to implement. Given the graph generated in the cause-effect approach, it is possible to write a program to implement the ENF algorithm and compute the matrix that is used to generate test cases. In summary, the ENF hardware approach has some merits as a base for a software method due to the fact that ENF algorithms and programs are available in the public domain, while cause-effect programs would have to be developed. Appendix C shows, for an example program, that the cause-effect approach and the ENF approach generate an identical set of test cases.

2.2.5 Mutation analysis

Mutation analysis is a method used to help the programmer generating the test cases develop a set of comprehensive test cases (Acree, 1980; DeMillo, 1980; Budd, 1980; DeMillo, 1983) This approach calls program, P, that is assumed to be correct or almost correct, to be modified to form programs P1,P2,...PN, that are each very similar to program P. These unique programs are called mutants of P and are generated by changing a statement or statements in P. Examples of the changes are: move the decimal point, reverse table dimensions, delete an instruction, substitute one variable name for another, and reversing the direction of a move instruction.

Test cases for P are then generated either informally or by using formal methods. The test cases are run against programs P,P1,P2,...PN and any of the P1,P2,...PN mutant programs that compute exactly the same results as P for all the test cases are considered active mutants. The other mutant programs differ from P by computing a different result for at least one of the test cases and they are, therefore, eliminated from further consideration. The next step is to generate additional test cases, that when run will differentiate between P and the set of active mutants. This can be accomplished by analyzing the instructions that were changed in the active set, an analysis that sometimes leads to the discovery of an error in the base program P.

Some of the active mutants may be functionally equivalent to the base program P (for example if the instruction in program P is A=(-B)**2 and the instruction in a mutant program is A=(+B)**2 then these two programs are equivalent and the mutant can be deleted from the active set). This process is repeated until all mutants are inactive or declared equivalent. The basic philosophy is that if the programs P1,P2,...PN are selected in an intelligent manner then the resulting test cases for P will be sufficient.

It has been shown (Acree, 1980) that it is not necessary to generate a mutant with more than one change from the base program (for example it is not required to change two instructions or make two modifications to one instruction). This is known as the coupling effect. He has also shown that the number of mutants required is proportional to the square of the number of lines of code. Any serious implementation efforts would require an automatic method of generating the mutant programs and testing for equivalence.

Another important concept of the mutation analysis paradigm is called the <u>competent programmer hypothesis</u> , which states that the program, P, is correct or nearly correct. If P is not correct then there is a high probability that one of the programs, P1, P2,... PN is correct. This hypothesis can be stated formally as follows: (Budd, 1980).

A competent programmer, after giving the task sufficient thought and pursuing the normal process of programming and debugging, has probably written a program that is either correct or "almost" correct, in that it differs from a correct program in simple ways.

The purpose of this hypothesis or assumption is to assure that a totally incorrect program is not shown, by way of mutation analysis, to be correct. For example, given a sort program and a set of test cases that separates the sort program from all its mutants, nothing is shown if the specifications of the program call for a square root program. Practically this is not a problem if every test case includes an expected result, as well as input parameters.

The concept of weak mutation testing has been developed (Howden, 1982). Mutation testing is modified in two ways, first given program P with a component C, then if one modifies C to form C' it is required that a test be developed to differentiate C' from C. It is not necessary for the test to differentiate P from P'. The second difference is that in the construction of C' only certain types of mutants are generated based on error studies (for example what types of errors are most prevalent). They include wrong relation operator, off by a constant, wrong coefficient, etc. There are two advantages of using weak mutation testing (versus mutation testing), there are fewer mutants due to the construction of limited types and it may be easier to generate the mutants. It is interesting that this method is similar to domain testing for testing of arithmetic relations, it is also similar, or a superset of, branch testing (a trivial example would have C' equal to the decision instruction).

2.2.6 Assertion Checking

Assertion checking involves the writing of special instructions called assertions that are then evaluated as the program is executed. These are usually tests for the range or values of input variables, internal variables and output variables. For example, a routine to compute the square root of x and put in a variable SQRTx could have an input assertion: ASSERT (X >= 0) and output assertion

ASSERT (SQRTx >= 0) and SQRTx * SQRTx = x). The use of assertions is a two step process; first the assertion statements are written and placed in the subject program; second, the program is executed using test cases developed using another method. These cases are independent of the number and content of the assertions, although a bad choice of test cases may cause the assertions to not be executed or not executed with effective input. Although it is possible that some simple assertions (for example ASSERT X is not modified) could be checked with a static analyzer, execution is usually required. It would be possible to write the assertions after the coding phase is complete, however, since a knowledge of the internal program logic is required, it makes more sense to write the assertions as the program code is written.

Assertion statements are usually additions to higher level languages and include an error reporting or stopping function when the conditions are not met. The languages should have the capability of being compiled with or without the assertion statements; in that way the extra overhead of executing these checks and the extra storage required could be avoided after the testing phase. Another approach would be to have a program switch to turn on and off the assertion overhead as required in the compiled program. This would save some of the execution time but none of the space.

Assertions can also `check the absolute variable range; relative variable ranges (for example ASSERT x=y); physical units of variables; loop invariant; maximum number of loop iterations, etc. There are several disadvantages of the assertion approach. First, there is not a generally accepted methodology to decide where to insert assertions, what they should check etc. It has been shown (Hiedler et al., 1982) that the number of assertions is not related to the Halstead or McCabe metric for program complexity. Programmer intuition and experience must be used to decide what assertions to include. Like any other instructions, these must be designed, tested, documented, debugged, etc. The second difficulty with the assertion approach is that it must be used in conjunction with another method to generate test cases. A third disadvantage is that in time dependent programs, the function of executing the assertions may modify the output.

2.2.7 Error seeding

Error seeding is discussed under the category of dynamic testing because it requires the program under test to be executed. This subject could also have been discussed in a section on reliability models; for it is, in fact, a reliability model that is used directly in dynamic testing (Mills, 1972). When a program is written and ready to be tested, errors are purposely inserted into the code by a party other than the tester. These seeded errors are then found during the testing phase, as are other indigenous errors. Based on the ratio of seeded to indigenous errors found, a prediction can be made on the remaining indigenous errors. One of the problems with this simple model is the assumption that all errors have the same probability of being found. Another related concern is the assumption that seeded errors are inserted randomly. Although this method appears rather intuitive in practice, it has not been used often and, with a few exceptions (Duran and Wiorkowski, 1978; Ramamoorthy and Bastani, 1982; Musa, 1980) is not discussed in the literature. In part, this is due to the seemingly unnatural operation of inserting known errors into a program that is supposed to be correct.

2.2.8 Symbolic execution

In all of the approaches discussed so far, except assertion checking, actual test data is generated and used as input to the program under test. A completely different approach, symbolic execution, requires no input data. Instead, the program is executed and whenever input is required, a symbolic parameter is inserted. For example, when the instruction READ DATA is encountered, DATA is assigned a symbolic value, say A. Later, if the instruction PUT DATA +1 INTO WORK is encountered, then WORK is set to A +1. In the case of a conditional branch, the parameters are kept in all directions, so a snapshot of the symbolic execution may say, for example, WORK is A +1 if X >= 0 and WORK is A if X < 0. After this approach is propagated through the program, the output can be expressed via symbols and logical operations. The user is then expected to manually inspect the output and correct any errors. For example, in a program to solve a quadratic equation for X, you would expect a symbolic output of the form:

$$X = \frac{-b + SQRT\ (b^2 - 4ac)}{2a} \text{ and } X = \frac{-b - SQRT\ (b^2 - 4ac)}{2a}$$

So, as can be seen, symbolic execution is not a typical test method since test data is not needed; however, it can be used in conjunction with other methods. A successful looking symbolic execution does not guarantee the program is reliable. In our quadratic equation example, under some conditions 4ac may cause an overflow and therefore, a program fault.

There are several problems with this approach. First, the output may be extremely complex and hard to manually recognize as the proper formula. For example, is X = -b/2a + (b * b - 4ac)1/2 /2a correct in the example above? The second problem is that of array references. It becomes very complex due to the fact that a program variable (an array element) may have a variable embedded, A(1) would be acceptable but A(DATA) where DATA is calculated adds complexity very quickly. Some of these problems have been partially solved as is discussed below.

An advantage of symbolic execution can be seen in mathematical or algorithmic type applications. It is not clear how these concepts would apply to more general type applications, say a data base update program.

CHAPTER 3, EFFECTIVENESS AND COMPLETENESS

3.1 PROBLEM DESCRIPTION

The problem is to develop a uniform measure of test case effectiveness. Some exist and are directly related to a test method, for example:

Branch Coverage:	% of Branches covered
Statement Coverage:	% of Statements covered
Error Seeding:	% of Errors Found
Mutation:	Number of Mutants Left
Path:	% of Paths Exercised

It should be noted that these measures are defined from zero percent to one hundred percent; however, the relationship of one test method to the others is not well understood from the above metrics. Some methods are not easy to measure, for example:

Assertions

Symbolic Execution

In this chapter a metric will be developed, with a range of zero to one, that can be used to help one evaluate the usefulness of various test cases. The significance of this metric is that prior to its development in this dissertation there was not a continuous measurement; as is discussed below, a metric did exist that was either one or zero. Another contribution is the idea that the metric developed can be used to help one evaluate the relative usefulness of a collection of test case sets. The approach is developed in general and specifically applied to six test approaches; however, it can be expanded to all test methods. In this way, there exists, for each test method, a common measurement scheme. For example, it will be possible to compare a set of test cases developed using statement coverage criteria with a set of test cases developed using cause-effect graphing.

The purpose of running test cases is to help demonstrate that a program is correct; that is, the output meets the specifications when operating under permitted conditions. The metric developed helps us to make an objective decision on whether the program is correct, instead of the more subjective decisions that are usually made.

In the following section we will define effectiveness for functions, expand the definition to completeness of programs and then expand these definitions to the range of zero to one, instead of binary values zero or one.

3.1.1 Effectiveness definition

Given a function f, f: D->R, with

D = domain(f)

R = codomain(f)

Range(f) is the set of all y in R such that
there exists an x in D with y = f(x)

Let <u>Sf</u> be defined as all f' with domain(f') = D

If T is a subset of D and Sf is a subset of <u>Sf</u> then T is <u>effective</u> for f relative to Sf, if and only if, T≠0 and for all f' in Sf, f'(T) = f(T) implies f' =f.

This is based on the definition in the literature (Howden, 1982) but restated to make it clearer for use in this paper. This concept is the same as the equal transformation concept in mathematical algebra.

For example, let $f(X) = X^2$

$$Sf = (X^2,\ X^3,\ 3X - 2)$$

The results of these functions can be seen in Figure 4 on page 53.

Now we assume that the domain of Sf = (0,1,2,3) and let T = (1). The T is not effective for f relative to Sf since $X^2 = X^3$ over T and $X^2 \neq X^3$ over the entire domain of Sf (for example for X = 2).

	Input		
	1	2	3
X^2	1	4	9
X^3	1	8	27
$3X - 2$	1	4	7

Figure 4. Effectiveness example

The test set T = (1, 2) is still not effective since for f' = 3X - 2, f'(1) = f(1) = 1 and f'(2) = f(2) = 4 and there exists an X, say X = 3 such that $f'(3) \neq f(3)$. The test set T = (1, 2, 3) is effective for f relative to

$$Sf = (X^2, X^3, 3X - 2);$$

however, since the set of all f' in Sf with f=f' over T is empty. That is to say the test set (1 , 2, 3) will differentiate X^2 from X^3 and 3X - 2. It should be noted that it may not be effective if another function is added to Sf say:

$$(X^2 -1) (X^2 - 4) (X^2 - 9) + X^2$$

since f'(X) = f(X) for all X in T for this function and there is an X, 0, where f(X)≠f'(X)

3.1.2 Statement coverage effectiveness

Now for a little more complicated example, let f(x) = P(x) where P is the program shown in Figure 5 on page 55. That is, the function f is that which is generated by P, or f(x) = P(x) for all x in the domain of f. The domain is (0,1,2,3) and the codomain of f is (0,1,2). This means that for each x in (0,1,2,3) there exists a y in (0,1,2), such that P(x)=y.

That is, f is the function computed by the program P. Let

Sf = (P1', P2', . . . P8')

where the Pn' are defined as the program formed by replacing instruction N in program P with the following:

```
1    IF A = 1  GOTO 5
2    IF A = 2  GOTO 7
3    B = 0
4    GOTO 8
5    B = 1
6    GOTO 8
7    B = A
8    END
```

Figure 5.   Statement coverage example

```
n       B='ABEND';END
```

The results of these functions can be see in Figure 6 on page 56.

Now the test set T = (1) is not effective for f relative to Sf since for A = 1, the instructions of P numbered 1, 5, 6 are executed so P1'(1), P5'(1), P6'(1) are not equal to P(1); however P2'(1), P3'(1), P4'(1), P7'(1) and P8'(1) are all equal to P(1) and they are not equal over the entire domain of Sf. If T = (1, 2) then P(x) = P3'(x) = P4'(x) for x = (1, 2) and they are not equal over the entire domain so T = (1, 2) is also not effective. It can be seen that T = (0, 1, 2)

B	Input		
	1	2	0
P	1	2	0
P1'	ABEND	2	0
P2'	1	ABEND	ABEND
P3'	1	2	ABEND
P4'	1	2	ABEND
P5'	ABEND	2	ABEND
P6'	ABEND	2	0
P7'	1	ABEND	0
P8'	1	ABEND	0

Figure 6. Statement effectiveness example

differentiates all the P' from P so the set T = (0, 1, 2) is effective for f(x) = P(x) with respect to the P's. The reason it is effective is that for any P', P is not equal to that P' over the entire set T. In fact, the 0 added to T could be any real ≠ 1, 2.

The function computed by P is

$m = 1$	$f(m) = 1$
$m = 2$	$f(m) = 2$
$m \neq (1,2)$	$f(m) = 0$

It can be seen that any set T that is effective is also a set that will result in statement coverage, any set that is not effective will not result in statement coverage. By defining Sf in this manner it can be seen that statement coverage can be stated in terms of effectiveness. The concept of completeness, defined in the next paragraph will eliminate the awkwardness of dealing with functions and programs that implement those functions.

3.1.3 Completeness definition

Given a program P, let

```
        D = domain(P) , that is all x where P(x)
                        is defined

        R = codomain(P)

        Range(P) is all y in R such that, there exists
                       an x in D such that y = P(x)
```

Let f be the function corresponding to P, that is f: D->R and for x in D, f(x) = P(x). We say f<->P. Given program, P, and programs, <u>P</u> = (P1, P2,...Pn) where each Pi has the same domain, D; and functions, <u>F</u> = (f1,f2,...fn) defined by f<->P. Then if T is a subset of D, then T is <u>complete</u> for P relative to <u>P</u> if it is effective for f relative to <u>F</u> .

This is a simplification and restatement of Howden's definition. In our previous example T = (0, 1, 2) is a complete set of test cases for P relative to the P's. We can say that T is complete relative to P1',P2',...P8' or equivalently T provides statement coverage.

<u>3.1.4 Branch coverage completeness</u>

Again if f(x) = P(x), let the set of P' be formed such that for each branch N in P there are two P's. One, Pn', that will result in a different output (from P) when branch N is taken (and the same output when branch N is not taken), and one, Pn", that will result in a different output (from P) when branch N is not taken (and the same output when branch N is taken).

for example if P is:

```
1    B = 0
2    IF A = 1  GOTO 4
3    B = 1
4    END
```

We can define P ' as:

```
1    B = 0
2    IF A = 1 THEN
        B = ABEND
3    B = 1
4    END
```

and P " as:

```
1    B = 0
2    IF NOT (A=1) THEN
            B = ABEND
3    IF (A=1) GOTO 5
4    B = 1
5    END
```

The functions defined area as follows:

P

A	B
1	0
≠1	1

P '

A	B
1	ABEND
≠1	1

P "

A	B
1	0
≠1	ABEND

So if T = (0), we have

```
P(0)     = 1

P'(0)    = 1

P''(0)   = ABEND
```

and since P'(0) = P(0), T = (0) is not complete. T = (1) is also not complete since

P(1)	= 0
P'(1)	= ABEND
P''(1)	= 0

that is P''(1) = P(1), The test set T = (0, 1) is complete since

P(0)	= 1	P(1)	= 0
P'(0)	= 1	P'(1)	= ABEND
P''(0)	= ABEND	P''(1)	= 0

That is, P is always differentiated from P' and P''. We see that the set where P(x) = P'(x) for T is empty, the same is true of P(x) = P"(x).

It is obvious that the set (0,1) provides branch coverage. By defining Sf in this manner it can be seen that branch coverage can be stated in terms of completeness. As a simple extension of branch coverage, multiple condition coverage can also be stated in terms of completeness.

3.1.5 Mutation analysis completeness

As we have seen for different testing approaches the challenge is to develop the set of P' in a way that will lead to our intuitive understanding of T. For mutation this is a straightforward, the P' set is the set of mutations of P.

If for a test set T, there exists a P' such P'(x) = P(x) for all x in T then either they are equivalent (in mutation parlance, P' dies) or the test set is not complete.

3.1.6 Error seeding completeness

Given the program P let the Pn' be formed such that each Pn' differs from P by one error that is intentionally inserted. If for a test case set T, P and one of the P's get the same result, since they are constructed to be different, then T is not complete. Once they produce different output, the seeded error is discovered. Looking at it this way, error seeding is a special case of mutation; however, in the error seeding approach the length of time it takes to find errors and the number of test cases run is used to predict the remaining number of indigenous errors.

3.1.7 Assertion completeness

Assertions don't really fit into this scheme since the assertions are added to code and then exercised with test cases that are developed using another approach. At a minimum we could assure that each assertion is at least exercised once using a simplified version of the P's for statement coverage.

3.1.8 Weak mutation completeness

In weak mutations testing a program P is segmented into sections C1,C2,...Cn. A mutation transformation is applied to a given Ci to produce $\underline{Ci}$ = (Ci', Ci", Ci'" ...). The test cases must then differentiate Ci from $\underline{Ci}$. That is, Ci must compute a different value from each of the $\underline{Ci}$ although, the program P with Ci and a version with one of the $\underline{Ci}$ may result in the same output.

Part of Howden's complex definition of completeness is based on the need to support this segmented program. A simpler approach would be to consider T to be complete for P relative to $\underline{P}$, if T is complete for Ci relative to $\underline{Ci}$ for all i <= n. $\underline{P}$ is the collection of programs formed from the Ci's.

3.2 AN EXTENSION OF THE COMPLETENESS MEASURE

The purpose of this section is to develop a metric to measure software correctness. Howden's completeness metric is used as a base. For each testing methodology, M and each set of test cases, T, and program, P, and variations of the program P = (P1,P2,... PN), T is either complete (1) or not complete (0) relative to M and P. First, an extension of completeness is defined to extend the possible number of values from just 0 and 1 to all the reals in the range 0 to 1. This will be based on the set P and our intuition about what should be tested. The resulting metric is applicable to both structural based testing approaches (for example, statement coverage) and functional based testing approaches (for example, cause-effect graphing).

Given a set of testing methods, M = (M1,M2,...Mn); a program P; a set of programs P = (P1,P2,...Pn) with the same domain as P; and a set T of test cases, then CA, the composite completeness metric is defined as:

$$CA(T,P,\underline{P}) = C(M1,T,P,\underline{P}) + \ldots C(Mn,T,P,\underline{P}) \text{ or}$$

$$CA(T,P,\underline{P}) = \sum_{r=1}^{n} C(Mr,T,P,\underline{P})$$

It should be noted that CA is based on a specific test set, T, a specific program, P, and the P programs. Since it is usually clear from the context, CA(T,P, P) is abbreviated to CA in the remainder of this paper. For the same reason, C(Mr) has been used and will continue to be used as an abbreviation for C(Mr,T,P, P).

Example programs are taken from the literature to show that the higher the CA the less likely that errors will occur. Then we can answer such questions as whether another test approach would help, should we work to raise one of the C's to one, etc. The time and effort of doing this is not included, that is we are assumed to have unlimited time, not a real world situation.

Given P = (P1,P2,...Pn); program P; and test set T, the meaning of complete with a range of 0 or 1 has been defined. It will now be expanded to the range 0 to 1 by the following extended definition of completeness.

If T is complete for P relative to m of the elements of P̲ then T is (m/n) * 100 percent complete or m/n complete.

For example if T is complete on all P̲ then the value is one. If T is not complete for any of the elements of P̲ then the value is zero. If T is complete for 25% of the elements of P̲ then value is .25. By using this method we have extended the definition of complete from just 0 and 1 to the range 0 to 1.

Statement coverage, as we discussed in an earlier section can be thought of as having the set P̲ defined by individually changing each instruction (that is one element of P̲ for each instruction). Therefore, the complete metric is a representation of the percent of instructions that are executed by the test set T, one hundred percent statement coverage results in a complete metric of one. This is represented as C(Statement) = 1. If no statements are executed C(Statement) = 0, if 25% are executed C(Statement) = .25.

For branch coverage, assuming each branch instruction has two possible outcomes, we can consider the P as having two members for each branch (see an earlier section, for details on how to generate P for branch coverage). So C(Branch) is one if we have 100% branch coverage, C(Branch) is .75 if we have 75% branch coverage etc. It is straightforward to expand the ideas used in the branch approach to develop C(Multiple Condition) such that it is one if and only if there is 100% multiple condition coverage.

In the cause-effect graphing approach a procedure is used to develop a cause-effect graph, which is used to generate a table from which we obtain the number of test cases and attributes of each test case. The elements of P can be constructed in the following manner. If T, the test set, is composed of test inputs, T1, T2,... Tm then a Pq is developed to correspond to each Tq and having the property that, when all the attributes that define Tq are true then P and Pq when exercised by Tq will give different results. In addition, when all the attributes that define Tq are not true then P and Pq when exercised by Tq will give the same results. In this way, we have defined the

C(Cause-effect) such that it represents the percent of the test cases required that are exercised. It should be noted that C(Cause-effect) is objectively obtained from the cause and effect conditions that are subjectively chosen by the person testing the code. This is not the case with the branch, statement and multiple condition metrics since they are based on the structure of the program and not the functions to be performed.

For path testing an approach similar to the cause-effect generation of P̲ is used. For each path the elements of P̲ can be built as follows. If Tq is a test case that exercised path q, then Pq should be formed such that Pq should produce the same result when path q is not exercised. In this way C(Path) is a measure of the paths that are exercised.

Mutation testing can be measured directly by the completeness metric. C(Mutation) is simply the percent of mutant programs, in the set, P̲ , that are eliminated by the mutation algorithm. That is C(Mutation) is defined as:

(Number of dead mutants)/(Total number of Mutants).

This concept can also be extended to other approaches, for example, in the error seeding methods, C(Error seeding) is:

(Number of found seeded errors)/(Total number of seeded errors).

As in mutation and cause-effect graphing, we must let common sense prevail and select a reasonable number of seeded errors in an intelligent manner. For example, if a small number of errors are seeded (say one) and if it is found, the C value of one could lead to a false sense of assurance. Assertions, domain testing, weak mutation and all other testing methods can also be measured in this manner.

The next step is to generate a composite completeness metric, CA, as the sum of the completeness metric for the approaches used. The methods to be studied are statement coverage, branch coverage, multiple condition coverage, path analysis, cause-effect graphing and mutation analysis. In this case, applying the concepts covered in section " 3.2 An extension of the completeness measure" on page 64:

```
CA = C(Statement) + C(Branch) +
     C(Multiple Condition) + C(Path) +
     C(Cause-effect) + C(Mutation)
```

The problems chosen to compute the composite completeness, CA, for are a text reformatter program, triangle classification program, a quadratic equation program and a sort program. Each is discussed in Chapter 4.

3.3 TEST METHODS AS A SUBSET OF MUTATION

One of the problems in dealing with test case and test set evaluation is the various different methods used to generate the test cases. It would be ideal if there were only one method of generating test cases and one simple approach to measuring the effectiveness of these test cases. As is explained below, it is possible to consider most testing approaches as a subset of mutation. As far as can be determined, this has not previously been reported in the literature. This result is not specifically used further in this dissertation; however, it serves to link the individual approaches discussed earlier in this chapter.

We will show in this section that every test approach that results in the generation of test cases can be considered as a subset of the mutation approach. Those that do not generate test cases such as program proving and symbolic execution are not considered in this discussion. For a given criteria and a program, P, a set of test cases T = (T1, T2, ...Tn) are needed; from this set T we will construct a set P̲ = (P1, P2, ... Pm) of mutants of P. We will then show that if T is complete relative to P̲ then T satisfies the criteria for the testing method that is being applied.

To construct the elements of Pq of P̲ , for each unique Tq in T, we generate a mutant Pq with the following properties:

1) Pq, when exercised by Tq, will produce a different result than P when exercised by Pq.

2) Pr, for $r \neq q$, when exercised by Tq will produce the same result as P when exercised by Pq.

This is certainly possible, for at a minimum, we could add a section to the beginning of program Pq to check for the input that is associated with test case Tq and, if present, generate an output outside the range of output of P. If not present, the same code as in P would be executed.

Now if T is complete for P relative to $\underline{P}$, then the criteria for this testing method is met. By construction the set T is complete relative to $\underline{P}$ for the program P.

It has been shown that all methods that generate test cases can be considered a mutation based approach by selecting the mutant programs based on the testing criteria. We have already covered how to do this algorithmically for statement coverage, branch coverage, multiple condition coverage, path testing and cause-effect graphing. Although this is an interesting observation, it is intuitively easier to think of statement coverage as exercising every instruction in a program versus differentiating a collection of mutants from a parent program. The same is true of the other methods discussed.

CHAPTER 4, EXAMPLES

4.1 TEXT REFORMATTER EXAMPLE

As a first example six methods of test case selection are considered: statement coverage, branch coverage, multiple condition coverage, path testing, cause-effect graphing and mutation, and a completeness value C is computed for each. The composite completeness measure CA is then computed as the summation of the C's. The text reformatter program (Goodenough and Gerhart, 1975) slightly modified (Walsh, 1983) is listed in Figure 7 on page 75. The Nassi-Shneiderman chart for this program is shown in Appendix E.

The problem can be stated as follows: Given an input text having the following properties:

I1: It is a stream of characters, where the characters are classified as break and nonbreak characters. A break

```
ALARM:=FALSE;
BUFPOS:= 0;
FILL:=0;
REPEAT
     INCHARACTER(CW)
     IF CW=BL OR CW=NL OR CW = ET
     THEN BEGIN
          IF BUFPOS≠0
          THEN BEGIN
               IF FILL+BUFPOS<MAXPOS AND FILL≠0
               THEN BEGIN
                    OUTCHARACTER(BL);
                    FILL:=FILL+1;END
               ELSE BEGIN
                    OUTCHARACTER(NL);
                    FILL:=0;END
               FOR K:=1 STEP 1 UNTIL BUFPOS DO
                    OUTCHARACTER(BUFFER(K));
                    FILL:=FILL+BUFPOS;
                    BUFPOS:=0;END END
     ELSE
          IF BUFPOS=MAXPOS
          THEN ALARM:=TRUE
          ELSE BEGIN
               BUFPOS:=BUFPOS+1;
               BUFFER(BUFPOS):=CW END
UNTIL ALARM OR CW=ET;
```

Figure 7.   Text Reformatter Program

character is a BL (blank), NL (new line indicator), or ET (end-of-text indicator).

I2: The final character in the text is ET.

I3: A _word_ is a nonempty sequence of nonbreak characters.

I4: A _break_ is a sequence of one or more break characters.

(As a result, the input can be viewed as a sequence of words separated by breaks with possibly leading and trailing breaks, and ending with ET.)

The program's output should be the same sequence of words as in the input with the following properties:

O1: A new line should start only between words and at the beginning of the output text, if any;

O2: A break in the input is reduced to a single break character in the output;

O3: As many words as possible should be placed on each line (that is, between successive NL characters);

O4: No line may contain more than MAXPOS characters (words and BL's);

O5: An oversize word (that is, a word containing more than MAXPOS characters) should cause an error exit from the program (that is, a variable Alarm should have the value TRUE);

We will assume throughout this example that MAXPOS = 3 so that short test cases can be developed. For C(Statement) to be 1, three test cases are needed, the following will suffice:

1. A, A, A, A
2. A, A, BL, B, B, ET
3. A, BL, B, ET

This will be referred to as test set I. This set doesn't result in all branches being exercised in all directions and C(Branch) is 7/8 (7 of 8 branch directions are exercised). The eight possible branch conditions can be seen in the Nassi-Shneiderman chart in Appendix E. This test set also doesn't exercise all the conditions as required in multiple condition coverage; 10 of 11 are exercised so C(Multiple condition) is 10/11.

We can add a test case:

4. NL, ET

to set I to obtain test set II which has the same characteristics as test set I except C(Branch) = 1. Likewise we can add test case:

5. A, A, A, ET

to set II to form set III which has the same characteristics of test set II except C(Multiple Condition) = 1.

The test set shown in Figure 8 on page 79 was developed to meet the criteria for the cause-effect graphing approach. Details of the cause-effect deviation are included in Appendix C.

Of course, C(Cause-effect graphing) is 1; and since the branch and statement test cases are a subset of IV, C(Branch) and C(Statement) are also 1; C(Multiple condition coverage) is again 10/11. Test set I covers 3

```
 1. A, A, A, A,
 2. A, A, BL, B, B, ET
 3. A, BL, B, ET
 4. NL, ET
 6. A
 7. A, BL, B, NL, ET
 8. A, BL, B, BL, ET
 9. BL, ET
10. ET
11. A, NL, ET
12. A, BL, ET
13. A, ET
14. A, A, BL, B, B, NL, ET
15. A, A, BL, B, B, BL, ET
```

Figure 8. Test'cases for C-E approach

of the 14 conditions, test set II and III each 4 of the 14 conditions.

For the mutation approach, a set of mutants are developed in an intuitive manner, that is, each instruction is deleted, equals are changed to not equals, less than to less than or equal to, less than to greater than, and to or, zero to ones, etc. Test set V is composed of test set IV plus the following two cases:

```
16. A, A, BL, B, NL, ET
17. A, A, BL, B, B, BL, C, BL, D, ET
```

There are about sixty mutant programs that were defined by modifying the original program. Those mutant programs that were syntactically correct were then exercised by the test cases in test set IV. Fifty eight of the sixty mutants were eliminated by this set of test cases. A small program was written to exercise the mutant programs and compare the actual and expected results.

For test sets I, II and III the most direct method of computing C(Mutation) is to exercise those test cases against the 60 mutants and determine how many are eliminated. Since, this is a cumbersome approach, a good approximation is the number of test cases included in the 16 required for a C(Mutation) = 1 with test case V. This is 3/16, 4/16, 4/16, respectively for test sets I, II, and III.

A final test set VI can be formed by adding:

5 A, A, A, ET

to test set V. This provides a [illegible] value of 1 for all the methods used.

Path testing requires that each path through a program be executed by at least one test case. For programs containing loops this requirement is impractical due to the very large or infinite number of paths that are possible. The paths are usually divided into a finite and manageable number of classes and at least one test case is generated for each class.

The method chosen to limit the number of paths in the text reformatter program is to continue with the assumption that the maximum line length is three and with the view of the program as handling one character at a time. If we consider that, the program can have four initial states, (zero to three characters in the buffer) then there are eighteen possible paths through the program. They can be enumerated as shown in Figure 9 on page 82.

Two paths are logically impossible and four more are impossible with the assumption that the maximum line

CONDITION	C-E CAUSE	PATH 111111111 123456789012345678
CW=BL or CW=NL or CW=ET	NOT 1	000011111111111111
BUFPOS=MAXPOS	2	0011--------------
BUFPOS≠0	6	----00111111111111
FILL+BUFPOS<MAXPOS AND FILL≠0	5 AND 7	------000000111111
INITIAL BUFPOS=1	NA	------100100100100
INITIAL BUFPOS=2	NA	------010010010010
INITIAL BUFPOS=3	NA	------001001001001
CW=ET OR ALARM=TRUE	8 OR NA	010101000111000111

Figure 9. Paths for text reformatter

length is three. The remaining twelve paths can be covered by test set VII, which includes the test cases 1, 3-8, 10, and 13-15 from set IV, (there are only eleven cases due to multiple loops with an individual test case). A summary of the test cases included in each set is shown in Figure 10 on page 83.

As we expected, the sum of the C's under consideration, CA, increases from test set I to VI. A summary is shown in Figure 11 on page 84.

TEST CASE	TEST SET I	II	III	IV	V	VI	VII
1	x	x	x	x	x	x	x
2	x	x	x	x	x	x	
3	x	x	x	x	x	x	x
4		x	x	x	x	x	x
5			x			x	x
6				x	x	x	x
7				x	x	x	x
8				x	x	x	x
9				x	x	x	
10				x	x	x	x
11				x	x	x	
12				x	x	x	
13				x	x	x	x
14				x	x	x	x
15				x	x	x	x
16					x	x	
17					x	x	

Figure 10. Test cases versus sets for Text Reformatter

Throughout this dissertation, we will measure the number of structural errors in a program, not the number of domain errors that will cause an incorrect result. The prime reason for this approach is to prevent the imprecise measurements that would result by having, say, one structural error that is responsible for an infinite number of domain errors. The next step is to determine if by increasing the value of CA the program is more

	TEST SET						
	I	II	III	IV	V	VI	VII
Statement Coverage	1	1	1	1	1	1	1
Branch	7/8	1	1	1	1	1	1
Multiple Condition Coverage	10/11	10/11	1	10/11	10/11	1	1
Cause Effect Graphing	3/14	4/14	4/14	1	1	1	11/14
Mutation	3/16	3/16	3/16	58/60	1	1	14/16
Path Testing	3/12	4/12	5/12	11/12	11/12	1	1
CA	3.44	3.77	3.86	5.79	5.82	6.0	5.75

Figure 11. Completeness Metric for Text Reformatter

reliable. One method is to determine which of the five sample errors in the Goodenough paper would be found by each of the test sets. A program was written in BASIC to answer this question. The results are shown in Figure 12 on page 86.

None of the test sets will uncover one of the errors due to our selection of a maximum line length of three. The error that is not found is deleting line 13 of the program: FILL = FILL + 1. The variable FILL is then one less than it should be; however, no errant decisions are made, since the maximum number of characters on a line is just three and there are no combinations of space available on a line and space used on a line that will cause an incorrect branch by the decision instruction that has FILL as a operand. If we increase the maximum number of characters on a line to any number greater than three, this instruction would be significant. The errors found represent various classes of errors including: inappropriate path selection, missing path and missing action. We can see from the above table, that for this example the higher the value of CA the higher the number of errors found.

TEST SET	CA	NUMBER OF ERRORS FOUND	TOTAL NUMBER OF ERRORS
I	3.44	2	5
II	3.77	3	5
III	3.86	3	5
IV	5.79	3	5
VII	5.82	3	5
V	6.00	4	5
VI	5.75	4	5

Figure 12. CA versus errors for Text Reformatter

4.2 TRIANGLE CLASSIFICATION EXAMPLE

The next example is the triangle classification problem that can be stated as follows: Determine whether three integers representing three lengths constitute an equilateral, isosceles, or scalene triangle or cannot be the sides of any triangle. A simple basic program to solve this problem is shown in Figure 13 on page 87.

For C (Statement) to be 1, the following four test cases will suffice:

```
50 READ A, B, C
100 IF NOT (A<B+C) THEN GOTO 500
110 IF NOT (B<A+C) THEN GOTO 500
120 IF NOT (C<A+B) THEN GOTO 500
130 IF (A=B) AND (B=C) THEN GOTO 600
140 IF (A=B) AND (B≠C) THEN GOTO 700
150 IF (A≠B) AND (A=C) THEN GOTO 700
160 IF (A≠B) AND (A≠C) AND (B=C) THEN GOTO 700
170 IF (A≠B) AND (A≠C) AND (B≠C) THEN GOTO 800
500 PRINT "NOT A TRIANGLE": END
600 PRINT "EQUILATERAL TRIANGLE": END
700 PRINT "ISOSCELES TRIANGLE": END
800 PRINT "SCALENE TRIANGLE": END
```

Figure 13. Triangle classification program

1. 1, 1, 1
2. 2, 2, 3
3. 3, 4, 5
4. 1, 0, 0

For C(Branch) = 1 we need four additional test cases for the branch directions not already exercised, a sufficient set is as follows:

5. 0, 1, 0
6. 0, 0, 1
7. 2, 3, 2
8. 3, 2, 2

Since there are no complex decision points C(Multiple Condition) is always equal to C(Branch). Also since there are no loops in this program C(Path) is always equal to C(Branch).

The next approach to be considered for this program is the mutation concept. About forty mutant programs are defined as follows: replace all less than signs by less than or equal to, replace less than by greater than, replace equals by not equal, replace not equal by equal, replace A by B, and by deleting each instruction. Several of the resulting mutant programs are impossible due to conflicting conditions. When the remainder are run against the existing eight test cases, two mutants remain. The test cases that must be added to kill these two mutants are:

9. 6, 2, 4
10. 3, 2, 4

The next approach used was cause-effect graphing with the causes as follows:

1. $A \neq B$
2. $B \neq C$
3. $B=C$
4. $A=C$
5. $A<B+C$
6. $B<A+C$
7. $C<A+B$

The relationships were straightforward and the effects were a equilateral triangle, an isosceles triangle, a scalene triangle or no triangle. The analysis showed six test cases were required; a set that satisfies the criteria is 1, 2, 3, 4, 7 and 8.

A summary of the test sets is given in Figure 14 on page 90.

Six relatively simple errors (Myers, 1976) were then inserted in the program to see which test cases would detect them as shown in Figure 15 on page 91.

As we can see, the number of errors found increase with an increasing CA, the same number found with the last

	CASES 1 - 4	CASES 1 - 8	CASES 1 - 10	CASES 1-4,7,8
STATEMENT COVERAGE	1	1	1	1
BRANCH COVERAGE	.5	1	1	6/8
MULTIPLE CONDITION COVERAGE	.5	1	1	6/8
PATH COVERAGE	.5	1	1	6/8
CAUSE-EFFECT GRAPHING	4/6	1	1	1
MUTATION	27/40	38/40	1	6/10
CA	3.84	5.95	6.00	4.85

Figure 14. Completeness Metric for Triangle Program

two sets due to the small difference in the CA's and since they were relatively simple errors.

TEST CASES	CA	NUMBER OF ERRORS FOUND	TOTAL NUMBER OF ERRORS
1 - 4	3.84	4	6
1-4,7,8	4.85	5	6
1 - 8	5.95	6	6
1 - 10	6.00	6	6

Figure 15. CA versus errors for triangle problem

4.3 QUADRATIC EQUATION EXAMPLE

A program to solve a quadratic equation (Kernighan and Plauger, 1974) was slightly modified and run in Fortran. The problem is to solve the quadratic equation $AX^2 + BX + C = 0$, that is, to find the two roots, one root or give an indication that it is not solvable. The program is shown in Figure 16 on page 92.

```
        WRITE (*,102) A,B,C,
102     FORMAT ('OA=', F12.5,  B='F12.5' C='F12.5)
        IF (B.EQ.O.AND.C.EQ.O) GOTO 15
        IF (B.NE.O.AND.C.NE.O) GOTO 50
        IF (A) 30,20,30
15      IF (A.EQ.O) GOTO 9035
20      WRITE(*,9010)
9010    FORMAT ("OTRIVIAL CASE, TWO OR MORE ZEROS')
        RETURN
30      IF (C) 60,40,60
40      XA=B/A
        XB=0
        GOTO 100
50      IF (A.NE.O) GOTO 60
        XA=-C/B
        XB=0.0
        GOTO 100
60      Q=B*B-4.*A*C
        XX=-B/(2.*A)
        IF (Q) 80,70,80
70      XA=XX
        XB=XX
        GOTO 100
80      QA=ABS(Q)
        XS=SQRT(QA)/(2.*A)
        IF (Q) 110,110,90
90      XA=XX+XS
        XB=XX-XS
100     WRITE (*,9020) XA,XB
9020    FORMAT (5H X1= ,F12.5,3X,4HX2 = ,F12.5)
        RETURN
110     XA=XS
        XB=-XS
        WRITE(*,9030) XX,XA
9030    FORMAT (5H X1 = ,F12.5,2H + ,F12.5)
        WRITE (*,9031) XX,XB
9031    FORMAT (5H X2 = ,F12.5,2H + ,F12.5)
        RETURN
9035    WRITE (*,9036)
9036    FORMAT ('OA=0 PROGRAM STOPPED')
        RETURN
        END
```

Figure 16.   Quadratic Equation Program

This program, which is not structured, has several known errors that were included as found in Kernighan and Plauger. By trial and error it can be seen that statement coverage can be accomplished (that is C(Statement) = 1) by the following test cases:

1.	A = 0	B = 0	C = 0
2.	A = 0	B = 1	C = 1
3.	A = 0	B = 0	C = 1
4.	A = 1	B = 1	C = 0
5.	A = 1	B = 5	C = 4
6.	A = 1	B = 2	C = 1
7.	A = 1	B = 1	C = 1

For Branch coverage to be complete, C(Branch) = 1, one needs two additional test cases:

8.	A = 1	B = 0	C = 0
9.	A = 1	B = 0	C = 1

For C(Multiple Condition) = 1, one must insure that all the IF statements are exercised in all directions; this requires two additional test cases:

10.	A = -1	B = 0	C = 1
11.	A = 1	B = 0	C = -1

For the cause - effect graphing approach, the inputs were as follows:

1) $A = 0$
2) $B^2 - 4AC > 0$
3) $B^2 - 4AC = 0$
4) $B^2 - 4AC < 0$
5) $B = 0$
6) $C = 0$

The outputs were the proper roots, found by evaluating the normal solution: Root 1 = $(-B + \text{SQRT}(B^2 - 4AC))/2A$ and Root 2 = $(-B - \text{SQRT}(B^2 - 4AC))/2A$, with special cases to avoid division by zero, to handle imaginary numbers, and to handle single and no root solutions. The result is that for C(Cause - effect) = 1, it is sufficient to exercise the program with test cases 1,5,6 and 7. It should be noted that cause - effect graphing does not use the actual program structure in generating the criteria for test cases; they are entirely based on the functions to be performed and the evaluator's knowledge of the problem.

The next approach to be evaluated is path testing. Since there are no loops in this program, it is reasonable to obtain a C(Path) = 1. Careful and tedious analysis shows that there are eleven theoretical paths through the program and ten are logically possible. These ten paths can be covered by test cases 1 - 10.

For the mutation approach, five types of mutants were selected:

1) A replaced by B
2) XA replaced by XB
3) EQ replaced by NE
4) NE replaced by EQ
5) Each line deleted

This is a limited set (there are fifty-six total mutants) due to the practical problems of generating and exercising mutant programs without a tool. Test cases 1 - 11 were used as a base and all the mutants were found with these cases.

A summary of the coverage discussed is given in Figure 17 on page 96; some of the results, that are less than one, represent the percent of the test cases present (a C of 1 would represent 100%) rather than the percent of coverage. Since these two metrics should be relatively close, this is done to reduce the computational complexity.

There were seven errors identified in the Kernighan and Plauger program. The number of errors that would be

	Test Cases				
	1-7	1-9	1-11	1,5,6,7	1-10
Statement Coverage	1	1	1	4/7	1
Branch Coverage	14/16	1	1	4/9	1
Multiple Condition Coverage	9/12	10/12	1	4/11	11/12
Cause-Effect Graphing	1	1	1	1	1
Mutation	7/11	9/11	1	4/11	10/11
Path Testing	7/10	9/10	1	4/10	1
Total CA	4.95	5.54	6.0	3.13	5.81

Figure 17. Completeness metric for quadratic equation

discovered by the methods discussed and with the test cases selected are shown in Figure 18 on page 97.

As can be seen, the number of errors found is monotone nondecreasing with the composite completeness metric, CA. One error is not found by any of the test cases; this is due to lack of a structured programming approach and the

TEST CASES	CA	NUMBER OF ERRORS FOUND	TOTAL NUMBER OF ERRORS
1,5,6,7	3.13	2	7
1-7	4.95	5	7
1-9	5.54	6	7
1-10	5.81	6	7
1-11	6.00	6	7

Figure 18. CA versus errors for quadratic program

nature of the error. The program, as written, will produce an incorrect message: "Trivial case, two or more zeros" if A=0, B≠0 and C=0. The correct answer is that there is one root at zero.

It is interesting to note that running these eleven test cases against the corrected program in Kernighan and Plauger results in three cases failing due to two errors. Due to the nature of the problem and the ease at which the results can be checked, it is natural to try the assertion approach on the incorrect program. If an ending assertion states that for any roots found $AX^2 + BX + C = 0$ then two of the seven errors woulc have been found; if an additional assertion, that the roots not be equal was inserted an additional error would have been uncovered. This is due to the fact that some of the errors were the printing of incorrect error messages.

4.4 SORT EXAMPLE

In this simple example a sort fragment of a program (McCracken, 1974) is analyzed. The problem is to sort the vector A which has N elements. A program to accomplish this is shown in Figure 19 on page 99.

Statement coverage is relatively straightforward; for C(Statement) = 1 we must have one test case:

1. A=(3,2,1)

For C(Branch) to be 1 we must add a test case with increasing elements.

```
70 NM1=N-1
80 FOR I=1 to NM1
90 IPI.US1=I+1
100 FOR J=IPLUS1 to N
110 IF (A(I)<>A(J)) THEN GO TO 150
120 TEMP=A(I)
130 A(I)=A(J)
140 A(J)=TEMP
150 NEXT J
160 NEXT I
```

Figure 19. Sort Program

2. A=(1,2,3)

Since there are no multiple conditions in the program, C(Multiple Condition) is equal to C(Branch). For C(Path) to be 1 many additional test cases would need to be generated based on the value of N. In order to limit this, we will consider a subset of the paths that execute the outer loop a minimum number of times (one), a maximum (nine was chosen as a workable limit), and a number of loops in the middle (say five). The inner DO loop is executed based strictly on the number of outer loop iterations. For each path the IF statement should be executed in both directions. For this criteria, in addition to the test cases already developed we need:

3. A=(1,2,3,9,8,5,4,7,6)
4. A=(5,3,1,2,4)

For the mutation approach, several substitutions are made, changing only one item at a time. The mutants are as follows: replace I by J, replace J by I, replace + by -, replace - by +, replace 1 by 0, replace < by >, replace < by <= and, deleting each line. Exercising these mutant programs with the test cases developed so far results in five mutants (of twenty eight) that are still alive; four mutants are correct, although not efficient and one requires an additional test case to differentiate it from the original program:

5. A=(-1, -2, -3, -4, 10)

Due to the relatively simple causes (if the vector is not in nondecreasing order, sort) and effects (a sorted vector), the cause- effect approach is not very practical. If we assume, for the cause-effect analysis only, that the maximum number of elements is three, and all permutations of order are included, then in addition to test case 1 and 2 we need the following:

6. A=(1, 3, 2)
7. A=(2, 1, 3)
8. A=(2, 3, 1)
9. A= (3, 1, 2)

A summary of the test cases developed is shown in Figure 20 on page 102.

Next seven typical errors were introduced, they included loops off by one, inappropriate initialization and improper path selection. The nine test cases were executed to see which errors they detected with the results shown in Figure 21 on page 103.

As can be seen the number of errors found is monotone nondecreasing with the CA metric.

4.5 DISCUSSION OF EXAMPLES

The four examples that were written in two languages (Basic and Fortran) have shown that, for the sample

	TEST CASES				
	1	1,2	1-4	1-5	1, 2, 6-9
STATEMENT COVERAGE	1	1	1	1	1
BRANCH COVERAGE	.5	1	1	1	1
MULTIPLE CONDITION COVERAGE	.5	1	1	1	1
PATH TESTING	.25	.5	1	1	.5
MUTATION	.2	.4	.8	1	.4
CAUSE-EFFECT GRAPHING	1/6	2/6	2/6	2/6	1
CA	2.61	4.22	5.12	5.32	4.90

Figure 20. Completeness metric for sort program

programs, the percent of known errors that are found by a test set behaves in a monotone, nondecreasing manner when compared with the composite completeness measure, CA. The significance of this metric, CA, is due to the fact that

TEST CASES	CA	NUMBER OF FOUND ERRORS	TOTAL NUMBER OF ERRORS
1	2.61	4	7
1, 2	4.22	5	7
1, 2, 6 - 9	4.90	5	7
1 - 4	5.12	6	7
1 - 5	5.32	7	7

Figure 21. CA versus errors for sort program

now, for any test cases selected by a particular method or methods, we have a uniform measurement. This has the potential to be used to assess the relative usefulness of test case sets; to help decide how much more effort should be put into testing, and to help decide when to stop testing. Prior to the development of the CA metric, there was no uniform method of evaluating sets of test cases. The evaluation of testing methods is no longer purely subjective, it is more systematic and objective. For easy reference, a summary of the data collected in Chapter 4 is found in Appendix A.

The reason for selecting the six test approaches that compose CA are two. First, they represent both white box and black box testing; both common approaches (statement) and less known and used methods (mutation). Second, the approaches used were not extremely difficult to implement with tools that are easily available. Statement, branch, and multiple condition coverage are relatively straightforward and although another person testing the program could develop a different set of test cases, it is likely that they would be similar. For path testing, when all paths could not reasonably be executed, due to loop iterations, it is important to select the path criteria in an intuitive and reasonable manner. It is possible that another person would select different path constraints, develop different test cases to exercise them, and discover different errors. The metric CA for path testing was based on the number of paths that were in the subset selected, not on the total number of possible paths. This is due to the high number of possible paths that would result in low CA values and small differences between CA values.

For mutation, one again has to select the mutants in an intuitive manner and in sufficient numbers. For example, it makes little sense to generate a thousand mutants based on the first line of multiline program and none based on the other lines. For the example programs, the mutants were based on experience. Cause-effect graphing can also be influenced by the level of detail used to develop the inputs and outputs.

As an extension to this work, a different set of test approaches could be evaluated; ground rules could be developed for path, mutation and cause-effect graphing concepts and more complex programs could be analyzed.

CHAPTER 5, AN EXPERIMENT

5.1 OVERVIEW OF EXPERIMENTAL APPROACH

The purpose of this chapter is to gain some experimental results to lend support to the conclusion that the completeness metric, CA, does predict the percentage of errors that are found in the example programs and in the experimental programs. An outline of the steps to be taken is as follows:

1) A problem will be defined and given to the subjects. They will return the first pass of their program (that is after the first error free compile). Also to be returned is a list of errors that were found between the first pass and their final program. These errors will be considered the known errors. This is a reasonable approach since the text reformatter is used and any additional errors that are found, as a result of running the sets of test cases

already developed, will be added to the collection of known errors.

2) The first pass programs will then be tested by developing six sets of test cases each; these will be developed by insuring that each test approach has a completeness value, C, of one for at least one of the six sets of test cases.

3) Based on the data in the previous chapter, and the experimental data to be collected, explain how CA will predict the number of errors. The first step is to develop an equation:

$$\begin{aligned}\text{Error \%} = {} & X1 * C(\text{Statement}) + X2 * C(\text{Branch}) + \\ & X3 * C(\text{Multiple Condition}) + X4 * C(\text{Path}) + \\ & X5 * C(\text{Cause-Effect}) + X6 * C(\text{Mutation}) + \\ & X0\end{aligned}$$

This will be done based on running a regression analysis on the data obtained. This analysis will be interpreted for the sample programs and an approach given, albeit, not a statistical approach, for an interpretation for an arbitrary program.

5.2 EXPERIMENTAL PROBLEM DEFINITION

The problem used in this experiment, the text reformatter problem, discussed in Goodenough and Gerhart's paper was slightly modified. The modifications were to insure that the programs are subroutines with a standard input and output format and to make the program slightly easier to automatically verify. The specifications for this program are included in Appendix B.

5.3 RESULTS OF EXPERIMENT

Programs were obtained as discussed above and were tested to see which test cases discovered the known errors. In addition, any additional errors that were found were added to the list of known errors. There were several errors of this type, partly due to voluntary method of having the programs written. Minor insignificant, annoying errors were corrected before running the experiment; for example, one program asked for the maximum number of characters before each test case. The cause-effect test set was developed in section "4.1 Text reformatter example" on page 74; the other test sets were developed based on the structure of the programs. The results are shown in Figure 23 on page 111 and Figure 22 on page 110.

The experiment was performed by two people, both using the same specifications; they are referred to in the figures as Experiment 1 and Experiment 2. As can be seen in Figure 24 on page 112 the metric CA is, again, monotone nondecreasing versus the percent of errors discovered.

5.4 DEVELOPING A MODEL

The purpose of developing a model, in this case, is to obtain a statistical base to make intuitive arguments. It is not to develop a pure mathematical model. There are

	Test Cases			
	1,2	1 2,11	1-3,5,8,10, 11,13-15	1-16
Statement Coverage	1	1	1	1
Branch Coverage	2/3	1	1	1
Multiple Condition Coverage	2/3	1	1	1
Path Testing	2/10* 15/18	3/10* 15/18	15/18	8/10* 15/18
Cause-Effect Graphing	2/3	3/13	8/13	1
Total CA	2.63	3.48	4.44	4.65

Figure 22. Completeness Metric for Experiment 1

two reasons for this; first, the amount of data available is not sufficient, at this time, to develop a theoretical model and, second, it has not been shown that all the required statistical assumptions are true. For example, it has not been shown that the form of the model is linear, although, intuitively that is a reasonable assumption; it has not been shown that the regressor variables are

	Test Cases			
	1,2,11,13 15	1,2,11, 13,15,6	1-13	1,3,5,6,22 12-4,16
Statement Coverage	1	1	1	.6
Branch Coverage	5/6* 19/24	19/24	5/6* 19/24	4/6* 19/24
Multiple Condition Condition	5/6* 19/24	19/24	5/6* 19/24	4/6* 19/24
Path Testing	3/9	4/9	7/9	1.0
Cause-Effect Graphing	5/13	6/13	1.0	7/13
Total CA	3.01	3.51	4.07	3.17

Figure 23. Completeness Metric for Experiment 2

measured without error, although it is reasonable that they have no or small errors. A simple linear regression model is used instead of a more sophisticated approach, such as, multiple variable regression or prime factor analysis due to the limited amount of data available and the desire to use a simplified statistical approach to make intuitive judgements.

TEST CASES	CA	NUMBER OF ERRORS FOUND	TOTAL NUMBER OF ERRORS
Experiment 1			
1,2	2.63	1	3
1,2,11	3.48	1	3
1-3,5,8,10,11 13-15	4.44	2	3
1-16	4.65	3	3
Experiment 2			
1,2,11,13,15	3.01	1	2
1,2,11,13, 15,6	3.51	1	2
1-13	4.07	2	2
1,3,5,6 11-14,16	3.17	1	2

Figure 24. CA Versus Errors for Experiment

The first approach used was to test the composite metric, CA, for significance as a predictor of the percent of known errors found by using the Statistical Analysis System, SAS (SAS, 1982). The results show that the percent of errors can be predicted, for the programs studied, as shown in the following equation:

```
Error % = .03 + .15  * CA =
        = .03 + .15  * (C(Statement) + C (Branch) +
          C (Multiple Condition) + C (Path) +
          C (Cause - Effect) + C (Mutation))
```

Unfortunately this model only accounts for 58% of the variation in the percent of errors found. That is 58% of the total variation is attributed to the fit rather than left to residual error.

The next approach used was multiple regression analysis. The model shown below was the result, with those coefficients whose Probability > |T| value is greater than .05 omitted, as not significant contributors to the error percent. The Probability > |T| is the probability that a + statistic would obtain a greater absolute value than that observed given that the true parameter is zero. It is a generally accepted, statistical assumption that if this variable is greater than .05, the associated parameter estimate is not significant.

Error % = .92 * C (Statement) + .48 C (Path) + .05

This model accounts for 75% of the variance, and from a mathematical standpoint is the best that can be developed using the method of linear regression.

The model above is not intuitively appealing because it used only the metrics for statement and path testing. Since there is an imbedding relationship between C(Statement), C(Branch) and C(Multiple Condition) a model was developed to include this relationship by considering another variable:

(C Statement) +.2 * C(Branch) +.1 * +C(Multiple Condition)

The coefficients for C(Branch) and C(Multiple Condition) were selected in an arbitrary manner based on intuition and the desire to keep the total variable significant, for the programs studied. This parameter was based on the intuitive concept that the set of tests cases that provides statement coverage is a subset of the set that provides branch coverage, which in turn is a subset of the set that provides multiple condition coverage. This approach helps to alleviate the problem of counting C(Multiple Condition) as three (that is one for C(Statement), one for C(Branch) and one for C(Multiple Condition)); although that is the approach we used by computing CA as the sum of all the C's. With the above example a C(Multiple Condition) of one would contribute 1.3. The model developed, as discussed above, will account for 71% of the variance, which is close enough to the maximum possible (75%) to give us confidence that it is a reasonable prediction.

The resulting equation, again with only those coefficients that are significant, for the programs studied, is as follows:

```
Error % = .61 * (C(Statement) + .2 * C(Branch) +
          .1 * C(Multiple Condition)) +
          .46 * C(Path)
```

The SAS analysis used to develop this model is included in Appendix D.

5.5 RESULTS OBTAINED FROM THE MODEL

The model developed shows that it is possible to predict the percent of errors that will be found based on the completion metrics for individual testing approaches for the programs studied.

Given an arbitrary program P and a set of test cases T, the technique developed can be used to compute the metric called "error percent". This metric should not be interpreted as the percent of errors that will be found; it should be interpreted as an indicator, for the person doing the testing, of the relative usefulness of the the test cases. For example, a metric of .6 does not mean 60 percent of the errors will be found; it should be interpreted, that if test cases are added and the .6 does not significantly increase then perhaps an incorrect approach to adding test cases has been chosen. On the other hand, if by adding test cases, the .6 increases the person testing should feel that progress is being made. This interpretation is consistent with the literature on the components of the composite metric; one hundred percent statement coverage is considered better that fifty percent coverage, however, the number of errors left and therefor the program correctness cannot be determined from the statement metric.

The final model accounts for 71% of the variation, that is 29% is due to residual error and 71% is due to fit.

Although there is not formal rule, statistically, 71% is good; it is reasonable that an increase in the number of data points will cause this to increase slightly. In fact, as data points were added during the development of this dissertation, the amount of variation due to fit, generally increased. This percent is also called coefficient of determination.

CHAPTER 6, SUMMARY AND CONCLUSIONS

6.1 SUMMARY

In this dissertation an approach to defining a test case metric is presented; this approach is demonstrated to be valid by means of examples taken from the literature. It can be used to help assess the relative usefulness of a collection of test cases. This was accomplished by first developing a metric to measure the completeness of test cases developed by a particular testing approach. A previously existing concept was expanded to permit this metric to have a continuous range. The second step was to develop a metric which is a composite of those developed for specific approaches. This composite metric was monotone, nondecreasing with the percent of errors found. The third step was to use the components of the composite metric to statistically predict the percent of errors remaining, in the programs studied.

In addition, items of lesser significance include the equivalence of the ENF and cause-effect graphing approach; the realization that most test methods can be considered as a subset of mutation and a classification of testing strategies.

6.2 FUTURE RESEARCH DIRECTIONS

The six methods of test case selection were chosen to allow both black box and white box testing, with an eye towards ease of implementation, or at least not an impossible implementation. In the future, addition testing approaches could be analyzed in this manner; for example, domain testing, weak mutation testing and error seeding could be directly evaluated using this methodology. Other testing approaches, such as symbolic execution, could also be evaluated using this approach, provided a completeness metric is developed; this does not seem complex, it just has not been done yet. Some thought has to be given to the definition of one hundred percent test coverage for symbolic execution.

Automated software tools are needed to pursue the metrics developed in this dissertation any further. In an ideal system, the programmer would submit his program and a set of test cases with expected results. The system would compute the individual completeness metrics and the composite metric; also available would be aides to increase the metric, for example, a list of paths not executed. As cited in Chapter 2, some software tools are available for specific approaches, however, no composite tools were found.

As indicated in Chapter 4, some of the values for completeness are based on approximations, mostly due to the clerical nature of running mutation exercises. For example, if ten test cases provided one hundred percent coverage, then it was assumed that any five test cases would provide fifty percent test coverage. This assumption is intuitively correct; however, the results may be slightly off. With an automated system, this approximation would not be necessary. The selection and number of mutant programs was also limited by the manual time involved; in an automated system the number of mutants to be generated would be limited by the computer time available and would generally be much larger.

It has been assumed, and shown, for the examples analyzed, that the higher the CA metric, the more likely it is that all the known errors will be discovered. Again, this seems logical; however, a future study, with more than six test methods, could address the questions: How high a CA value is enough? When should you stop to obtain a given confidence that all the errors are found?

It has also been assumed that an error is an error; that is, there is not a severity metric for errors. This is clearly not the case. For example, a formatting error on an airline report is much less severe than an airline controller program error that causes a plane to crash. It seems feasible that the severity of errors could be defined and then a test case could be given a weighted value that it would contribute to the completeness measure.

The problem of the size of the domain of Sf in some cases is obvious. For example, with branch coverage, in other cases it is not obvious. In the case of mutation, one can develop a large number of mutants, insure the test cases differentiate them, and obtain a high completeness measure; however, if the mutants are not selected in an intelligent manner, this may not be significant. That is to say, in the future, a measurement should be developed to measure the largeness or goodness of the set Sf.

APPENDIX A, SUMMARY OF EXAMPLES

Abbreviations used in this appendix:

PGM	-	Program
TR	-	Text Reformatter Program
QUAD	-	Quadratic Equation Program
TRI	-	Triangle Program
SORT	-	Sort Program
STM	-	Statement Coverage
BR	-	Branch Coverage
MCC	-	Multiple Condition Coverage
CE	-	Cause-Effect Graphing
MUT	-	Mutation Testing
PATH	-	Path Analysis
CA	-	Composite Completeness Metric
ER%	-	Percent of Known Errors Found

PGM	ER%	STM	BR	MCC	CE	MUT	PATH	CA
TR	.4	1	.87	.91	.21	.19	.25	3.24
TR	.6	1	1	.91	.29	.19	.33	3.77
TR	.6	1	1	1	.29	.19	.92	3.86
TR	.6	1	1	.91	1	.97	.92	5.79
TR	.8	1	1	.91	1	1	.92	5.82
TR	.8	1	1	1	1	1	1	6
QUAD	.71	1	.87	.75	1	.63	.7	4.95
QUAD	.86	1	1	.83	1	.81	.9	5.54
QUAD	.86	1	1	1	1	1.0	1	6
QUAD	.28	.57	.44	.36	1	.36	.4	3.13
QUAD	.86	1	1	.92	1	.91	1	5.81
TRI	.67	1	.5	.5	.67	.67	.5	3.84
TRI	1	1	1	1	1	.95	1	5.96
TRI	1	1	1	1	1	1	1	6
TRI	.83	1	.75	.75	1	.6	.75	4.85

PGM	ER%	STM	BR	MCC	CE	MUT	PATH	CA
SORT	.57	1	.5	.5	.16	.2	.25	2.61
SORT	.71	1	1	1	.33	.4	.5	4.22
SORT	.85	1	1	1	.33	.8	1	5.12
SORT	1	1	1	1	.33	1	1	5.32
SORT	.71	1	1	1	1	.4	.5	4.90

APPENDIX B, SPECIFIACTIONS GIVEN TO SUBJECTS

B.1 TEXT REFORMATTER

The following is the specifications and examples given to subjects who wrote a text reformatting program:

Submit a program prior to it's being debugged and then after it is debugged, with some kind of indication of the errors that were found.

Assume the input is in a vector, called A, and the output is to be put into a vector called B. (that is, do not really print anything)

The problem can be stated as follows: Given an input text having the following properties:

I1: It is a stream of characters, where the characters are classified as break and nonbreak characters. A break character is a @ (blank), $ (new line indicator), or % (end-of-text indicator).

I2: The final character in the text is %.

I3: A word is a nonempty sequence of nonbreak characters.

I4: A break is a sequence of one or more break characters.

(As a result, the input can be viewed as a sequence of words separated by breaks with possibly leading and trailing breaks, and ending with %.)

The program's output should be the same sequence of words as in the input with the following properties:

O1: A new line should start only between words and at the beginning of the output text, if any;

O2: A break in the input is reduced to a single break character in the output;

03: As many words as possible should be placed on each line (i.e., between successive $ characters); Any $ in the input should not result in a $ in the output unless the line is full (i.e. a $ in the input can be considered just like a @ in the input)

04: No line may contain more than MAXPOS characters (words and @'s);

05: An oversize word (i.e., a word containing more than MAXPOS characters) should cause an error exit from the program (i.e., put "ERROR" in B.);

Some examples might be helpful. MAXPOS is the maximum number of characters that can be set to a constant at the beginning of your program. For the following examples we will assume that MAXPOS is set to five.

Example 1:

Input in A:

```
This@is@@a$test%
```

Output that logically would be printed:

```
this
is a
test
```

Output in B:

```
$this$is@a$test
```

Example 2:

Input in A:

```
a@bigword%
```

Output that logically would be printed:

```
"ERROR"
```

Output in B:

```
"ERROR"
```

Example 3:

Input in A:

```
@@@there@@@are$$a@lot@of@@thing@$@A$B$C$D%
```

Output that logically would be printed:

```
there
are a
lot
of
thing
A B C
D
```

Output in B:

```
$there$are@a$lot$of$thing$A@B@C$D
```

APPENDIX C, EQUIVALENCE ENF AND C-E PROCEDURES

C.1 SUMMARY OF APPROACH

The purpose of this appendix is to convince the reader that the cause-effect graph procedures and the Equivalent normal form (ENF) procedure can result in an identical set of test cases being developed. The significance of this equivalency is that the ENF algorithm is well documented and has been implemented for years in the hardware arena. The cause-effect graphing approach was developed in 1973, and although it appears to be a very logical approach to test case selection, it is not extensively used. This is partly due to the fact that once the graph is generated, there is not an easy-to-follow algorithm to generate the test cases. Since the ENF procedure is equivalent, the graph can be generated by way of the cause-effect method and then the test cases automatically produced by way of the ENF algorithm.

In the future, a procedure could be written, or possibly picked up from a hardware system to produce the ENF test conditions. This was not done in this dissertation due to the lack of extremely complex programs to be analyzed. This equivalence is shown by procedurally stepping through a program of medium complexity, the text reformatter program, discussed in an earlier section.

C.2 CAUSE-EFFECT FOR TEXT REFORMATTER

As the first step in cause-effect graphing, all the causes (input conditions or system transformations) are identified and given a unique identifying number. Next, all effects (output conditions or changes in the system state) are identified and numbered. A graph is then generated by linking the causes to the effects with the proper logical relationships. The relationships used are AND, OR, identity (straight line) and NOT as well as various constraint symbols. A parenthesis is used to indicate the scope of the AND's and OR's. For example, in

Figure 26 on page 136, input conditions 3 OR 4 OR 8 must be true in order for condition 20 to be true.

As the size and ability to work with the cause-effect graphs increases quickly as the program specifications become more complex, the graph generated for our sample problem will represent the processing of just one character. The processing of this character will be affected by the prior state of the program, for example, what characters if any, were processed prior to the current character. The input conditions (1-8) and output conditions (90-94) are shown in Figure 25 on page 134.

The resulting cause-effect graph, developed from the Nassi-Shneiderman chart, is shown in Figure 26 on page 136. The 0 and dotted line connecting input conditions 1, 3, 4, and 8 indicate that one and only one of these input conditions are possible at one time. Reading the graph indicates, for example, that if condition 3 is true (the character is a NL) then the intermediate condition 20 is true. If condition 6 is also true (at least one character has previously been found and not printed), then

Input Conditions:

1.	CHARACTER IS NOT BL OR NL OR ET	
2.	BUFPOS = MAXPOS	(word is too long)
3.	CHARACTER IS NL	
4.	CHARACTER IS BL	
5.	BUFPOS + FILL<MAXPOS	(word found will fit on current line)
6.	BUFPOS ≠ 0	(at least one character found and not printed)
7.	FILL ≠ 0	(a word was already printed on this line)
8.	CHARACTER IS ET	

Output conditions:

90.	NO OUTPUT	(character put in buffer)
91.	ALARM	(word size too long)
92.	BL AND BUFFER PRINTED	
93.	NL AND BUFFER PRINTED	
94.	NO OUTPUT	(multiple or preceding break character)

Figure 25. Input/output conditions for C-E graph

condition 21 is true (an intermediate state). If condition 22 is also true (the previous word was already printed on the line and the current word will fit on the same line), then output 92 is true. This means a BL will be printed followed by the word in the buffer.

The next step in using the cause-effect graphing methodology is to generate a limited entry decision table that represents the portions of the graph that makes each output condition true (one at a time). The method used is to sequentially select each effect to be true and to trace back through the graph to find all combinations of causes that will make the given effect true. Each combination of effects is recorded in the decision table as a row. Some possible combinations may be ignored as they are not all necessary to generate the test cases and there may be an unreasonable number or combinations. All possible combinations may in fact mask certain causes. As an example, if four conditions are ORED together, it is only necessary to iterate four input conditions to make this true (each input true, while the others are false) instead of the fifteen possible combinations that make the output true.

The decision table for the text reformatter problem is shown in Figure 27 on page 137. The rows represent the condition of each cause or effect while the columns represent a particular test case to be developed. A dash (-) indicates a don't care position. This decision table shows that we must develop fourteen test cases.

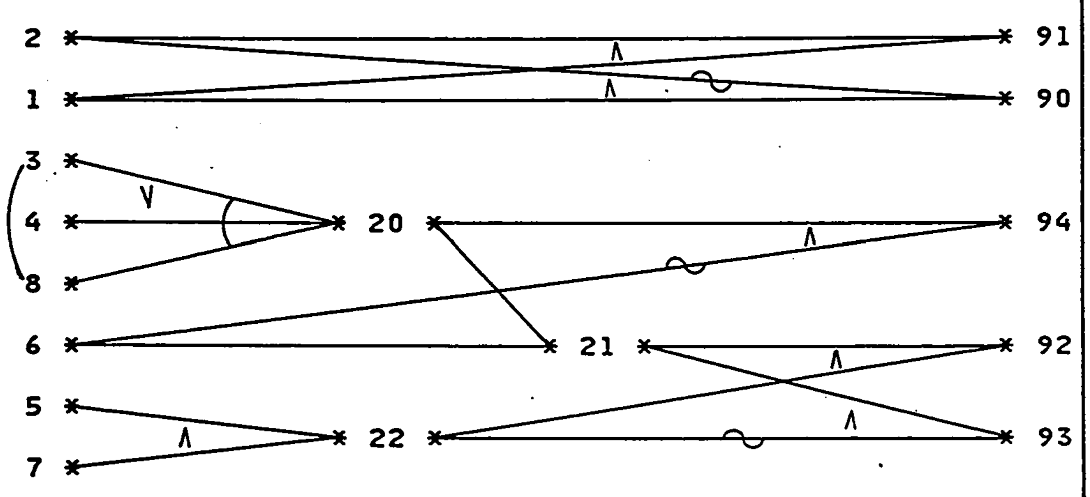


Figure 26.   Cause-effect graph for text reformatter

The final step is to convert the columns of the decision table into test cases. This is accomplished in a trial-and-error manner by inspecting the decision table and generating a test case for each column. For example, for column 1, we need condition 1 to be true (character is not a break character) and condition 2 (word is too long) to be false. The choice of the letter A as an input

TEST CASE NUMBER

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	1	-	-	-	-	-	-	-	-	-	-	-	-
2	0	1	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	1	0	0	1	0	0	1	0	0	1	0	0
4	-	-	0	1	0	0	1	0	0	1	0	0	1	0
5	-	-	1	1	1	-	-	-	1	1	1	0	0	0
6	-	-	1	1	1	0	0	0	1	1	1	1	1	1
7	-	-	1	1	1	-	-	-	0	0	0	1	1	1
8	-	-	0	0	1	0	0	1	0	0	1	0	0	1
90	1	0	0	0	0	0	0	0	0	0	0	0	0	0
91	0	1	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	1	1	1	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	1	1	1	1	1	1
94	0	0	0	0	0	1	1	1	0	0	0	0	0	0

Figure 27. Decision table for C-E method

obviously satisfies these constraints. One choice for the fourteen test cases, as well as the expected output is shown in Figure 28 on page 138.

Test case	Input	Expected output
1	A	(Character A in BUFFER)
2	A,A,A,A	ALARM MESSAGE
3	A,BL,B,NL,ET	A BL B
4	A,BL,B,BL,ET	A BL B
5	A,BL,B,ET	A BL B
6	NL,ET	(NO OUTPUT)
7	BL,ET	(NO OUTPUT)
8	ET	(NO OUTPUT)
9	A,NL,ET	A
10	A,BL,ET	A
11	A,ET	A
12	A,A,BL,B,B,NL,ET	AA BB
13	A,A,BL,B,B,BL,ET	AA BB
14	A,A,BL,B,B,ET	AA BB

Figure 28. Test cases for C-E graphing method

Cause-effect graphing is used as a procedural method to generate test cases. In addition, insight is gained into the problem to be solved by converting the specifications into the boolean graph. It can also assist in the discovery of incomplete and inconsistent specifications. Note that, generally, the test cases developed using the multiple condition coverage approach are not a subset of the cause-effect graph test cases. This situation is due to the fact that in multiple condition coverage all possible combinations of condition outcomes must be covered (for example, NOT 5 AND NOT 7 in Figure 26 on page 136). This is not the case with the cause-effect graph algorithm.

C.3 EQUIVALENT NORMAL FORM FOR TEXT REFORMATTER

This method to generate a set of test cases is based on the equivalent normal form (ENF) of a hardware circuit. The ENF is developed by expressing the output of each gate(output conditions or intermediate states for software) as a function of the inputs and at the same time preserving the identity of each gate.

The graph in Figure 26 on page 136 can be used as a basis for generating the ENF. The ENF's for our sample problem are shown in Figure 29 on page 141.

Each character (for example NOT 5[93]) is called a _term._ Terms connected by ANDS are called _literals._ The next step is to test each literal in an ENF for stuck at 0 by assigning 1's to all literals in the term containing it and making all other terms equal to zero. It is only necessary to test one literal per term using this method (testing more will result in duplicate tests). Applying these procedures we obtain the following result:

```
 for ENF (91) - 1[91]=1
                2[91]=1

 for ENF (90) - 1[90]=1
                NOT 2[90]=1, 2[90]=0
```

```
ENF(91) = (1 AND 2)[91] = 1[91] AND 2[91]

ENF(90) = (1 AND NOT 2)[90] = 1[90] AND NOT 2[90]

ENF(94) = (NOT 6 AND 20)[94] =
          (NOT 6 AND (3 OR 4 OR 8)[20])[94] =
          (NOT 6[94] AND 3[20,94]) OR
          (NOT 6[94] AND 4[20,94]) OR
          (NOT 6[94] AND 8[20,94])

ENF(92) = (22 AND 21)[92] =
          ((5 AND 7)[22]  AND (6 AND 20)[21])[92]=
          5[22,92] AND 7[22,92] AND
          6[21,92] AND 20[21,92]=
          (5[22,92] AND 7[22,92] AND
          6[21,92] AND 3[20,21,92] OR
          (5[22,92] AND 7[22,92] AND
          6[21,92] AND 4[20,21,92] OR
          (5[22,92] AND 7[22,92] AND
          6[21,92] AND 8[20,21,92]

ENF(93) = (NOT 22 AND 21)[93] =
          (NOT (5 OR 7)) [22,92] AND (6 AND 20)[21,92]
         =(NOT 5[22,92]OR NOT 7 [22,92]) AND 6[21,92]
          AND (3 OR 4 OR 8)[20,21,92]=
          NOT 5[22,92] AND 6[21,92] AND 3[20,21,92] OR
          NOT 7[22,92] AND 6[21,92] AND 3[20,21,92] OR
          NOT 5[22,92] AND 6[21,92] AND 4[20,21,92] OR
          NOT 7[22,92] AND 6[21,92] AND 4[20,21,92] OR
          NOT 5[22,92] AND 6[21,92] AND 8[20,21,92] OR
          NOT 7[22,92] AND 6[21,92] AND 8[20,21,92]
```

Figure 29.   ENF for example program

```
for ENF (94)

        - Literal 1 - NOT 6[94]=1, 6[94]=0

                      3[20,94]=1, 4[20,94]=0, 8[20,94]=0

          Literal 2 - 6[94]=0

                      4[20,94]=1, 3[20,94]=0, 8[20,94]=0

          Literal 3 - 6[94]=0

                      8[20,94]=1, 3[20,94]=0, 4[20,94]=0
```

Similar expressions can be developed for ENF's 92 and 93. A summary of the tests developed is shown in Figure 30 on page 143.

As the anticipated output for each ENF is 1 we see that these are exactly the same conditions that were generated using the cause-effect graphing method.

The ENF algorithm also requires that tests be generated for the stuck at 1 fault (a false output is expected) for all output conditions. This is not required

		Input Conditions							
		1	2	3	4	5	6	7	8
	91	1	1	-	-	-	-	-	-
	90	1	0	-	-	-	-	-	-
ENF	94	-	-	1	-	-	0	-	-
	94	-	-	-	1	-	0	-	-
	94	-	-	-	-	-	0	-	1
	92	-	-	0	0	1	1	1	1
	92	-	-	0	1	1	1	1	0
	92	-	-	1	0	1	1	1	0
	93	-	-	1	0	0	1	1	0
	93	-	-	1	0	1	1	0	0
	93	-	-	0	1	0	1	1	0
	93	-	-	0	1	1	1	0	0
	93	-	-	0	0	0	1	1	1
	93	-	-	0	0	1	1	0	1

Figure 30. ENF conditions for text reformatter

in this case since we tactfully chose output that are mutually exclusive.

APPENDIX D, STATISTICAL ANALYSIS

LIST OF DATA

OBS	ERROR PERCENT	Statement Coverage	Branch Coverage	Multiple Condition Coverage
1	0.40	1.00	0.87	0.91
2	0.60	1.00	1.00	0.91
3	0.60	1.00	1.00	1.00
4	0.60	1.00	1.00	0.91
5	0.80	1.00	1.00	0.91
6	0.80	1.00	1.00	1.00
7	0.71	1.00	0.87	0.75
8	0.86	1.00	1.00	0.83
9	0.86	1.00	1.00	1.00
10	0.28	0.57	0.44	0.36
11	0.86	1.00	1.00	0.92
12	0.67	1.00	0.50	0.50
13	1.00	1.00	1.00	1.00
14	1.00	1.00	1.00	1.00
15	0.83	1.00	0.75	0.75
16	0.57	1.00	0.50	0.50
17	0.71	1.00	1.00	1.00
18	0.85	1.00	1.00	1.00
19	1.00	1.00	1.00	1.00
20	0.71	1.00	1.00	1.00
21	0.50	1.00	0.65	0.65
22	0.50	1.00	0.79	0.79
23	1.00	1.00	0.65	0.65
24	0.50	0.6	0.52	0.52
25	0.33	1.0	0.66	0.66
26	0.33	1.0	1.00	1.00
27	0.66	1.0	1.00	1.00
28	1.00	1.0	1.00	1.00

LIST OF DATA, CONTINUED

OBS	Cause Effect Graphing	MUTATION TESTING	PATHS EXECUTED	VARIABLE TOTALS
1	0.21	0.19	0.25	3.44
2	0.29	0.19	0.33	3.77
3	0.29	0.19	0.42	3.86
4	1.00	0.97	0.92	5.79
5	1.00	1.00	0.92	5.82
6	1.00	1.00	1.00	6.00
7	1.00	0.63	0.70	4.95
8	1.00	0.81	0.90	5.54
9	1.00	1.00	1.00	6.00
10	1.00	0.36	0.40	3.13
11	1.00	0.91	1.00	5.81
12	0.67	0.67	0.50	3.84
13	1.00	0.95	1.00	5.96
14	1.00	1.00	1.00	6.00
15	1.00	0.60	0.75	4.85
16	0.16	0.20	0.25	2.61
17	0.33	0.40	0.50	4.22
18	0.33	0.80	1.00	5.12
19	0.33	1.00	1.00	5.32
20	1.00	0.40	0.50	4.90
21	0.38	0.00	0.33	3.01
22	0.49	0.00	0.44	3.51
23	1.00	0.00	0.77	4.07
24	0.53	0	1.00	3.17
25	0.15	0	0.16	2.63
26	0.23	0	0.25	3.48
27	0.61	0	0.83	4.44
28	1.00	0	0.65	4.65

PLOT OF ERRORS BY TOTAL OF C METRICS

```
    PLOT OF ERRORS*TOT

   1.0 +                         *       *         *      **
       |
       |
       |
   0.9 +
       |                                       *     *  *  *
       |                                    *
       |
E  0.8 +                                                 * *
R      |
R      |
O      |                           *        **
R  0.7 +
       |                      *
P      |                              *
E      |
R  0.6 +                     **                         *
C      |      *
E      |
N      |
T  0.5 +           * *    *
       |
       |
       |
   0.4 +                 *
       |
       |      *          *
       |
   0.3 +
       |             *
       |
       |
   0.2 +
       |
       ----+-------+-------+-------+-------+-------+-------+----
          2.4     3.0     3.6     4.2     4.8     5.4     6.0

                           VARIABLE TOTALS
```

Regression of Errors on Total of Other Variables

STATISTICS FOR CA

```
DEP VARIABLE: ERRORS   ERROR PERCENT

                          SUM OF         MEAN
SOURCE     DF            SQUARES       SQUARE         F VALUE

MODEL       1           0.754467     0.754467          36.228
ERROR      26           0.541458     0.020825
C TOTAL    27           1.295925
PROB > F                0.0001

    ROOT MSE            0.144310     R-SQUARE          0.5822
    DEP MEAN            0.697500     ADJ R-SQ          0.5661
    C.V.                20.68956

                       PARAMETER     STANDARD   T FOR H0:
VARIABLE   DF           ESTIMATE        ERROR  PARAMETER=0     PROB >
                                                                 |T|

INTERCEP    1           0.031398     0.113977        0.275     0.7851
TOT         1           0.148152     0.024614        6.019     0.0001
```

PREDICTED VERSUS ACTUAL FOR CA

OBS	ACTUAL	PREDICT VALUE	STD ERR PREDICT	RESIDUAL	STD ERR RESIDUAL	STUDENT RESIDUAL
1	0.400000	0.541041	0.037676	-.141041	0.139305	-1.012
2	0.600000	0.589931	0.032606	0.010069	0.140578	0.072
3	0.600000	0.603265	0.031446	-.003265	0.140842	-0.023
4	0.600000	0.889198	0.041930	-.289198	0.138084	-2.094
5	0.800000	0.893643	0.042493	-.093643	0.137912	-0.679
6	0.800000	0.920310	0.045979	-.120310	0.136789	-0.880
7	0.710000	0.764750	0.029472	-.054750	0.141268	-0.388
8	0.860000	0.852160	0.037470	0.007840	0.139360	0.056
9	0.860000	0.920310	0.045979	-.060310	0.136789	-0.441
10	0.280000	0.495114	0.043294	-.215114	0.137662	-1.563
11	0.860000	0.892161	0.042305	-.032161	0.137970	-0.233
12	0.670000	0.600302	0.031694	0.069698	0.140786	0.495
13	1.000	0.914384	0.045190	0.085616	0.137052	0.625
14	1.000	0.920310	0.045979	0.079690	0.136789	0.583
15	0.830000	0.749935	0.028630	0.080065	0.141441	0.566
16	0.570000	0.418075	0.053842	0.151925	0.133889	1.135
17	0.710000	0.656599	0.028106	0.053401	0.141546	0.377
18	0.850000	0.789936	0.031299	0.060064	0.140875	0.426
19	1.000	0.819567	0.033986	0.180433	0.140251	1.287
20	0.710000	0.757343	0.029028	-.047343	0.141360	-0.335
21	0.500000	0.477336	0.045626	0.022664	0.136907	0.166
22	0.500000	0.551412	0.036508	-.051412	0.139615	-0.368
23	1.000	0.634377	0.029219	0.365623	0.141321	2.587
24	0.500000	0.501040	0.042534	-.001040	0.137899	-0.008
25	0.330000	0.421038	0.053418	-.091038	0.134059	-0.679
26	0.330000	0.546967	0.037003	-.216967	0.139485	-1.555
27	0.660000	0.689193	0.027307	-.029193	0.141703	-0.206
28	1.000	0.720305	0.027534	0.279695	0.141659	1.974

STATISTICS FOR INDIVIDUAL METHODS

```
DEP VARIABLE: ERRORS   ERROR PERCENT

                         SUM OF          MEAN
SOURCE      DF          SQUARES        SQUARE          F VALUE

MODEL        4         0.924726      0.231182           14.324
ERROR       23         0.371199      0.016139
C TOTAL     27         1.295925
PROB > F               0.00001

    ROOT MSE           0.127040      R-SQUARE           0.7136
    DEP MEAN           0.697500      ADJ R-SQ           0.6637
    C.V.               18.21357

                      PARAMETER      STANDARD     T FOR H0:
VARIABLE    DF         ESTIMATE         ERROR   PARAMETER=0     PROB > |T|

INTERCEP     1        -0.431229      0.223855        -1.926         0.0665
St,Br,MCC    1         0.615599      0.174480         3.528         0.0018
CE           1         0.135203      0.091763         1.473         0.1542
MUT          1        -0.058904      0.090538        -0.651         0.5218
PATH         1         0.461587      0.127343         3.625         0.0014
```

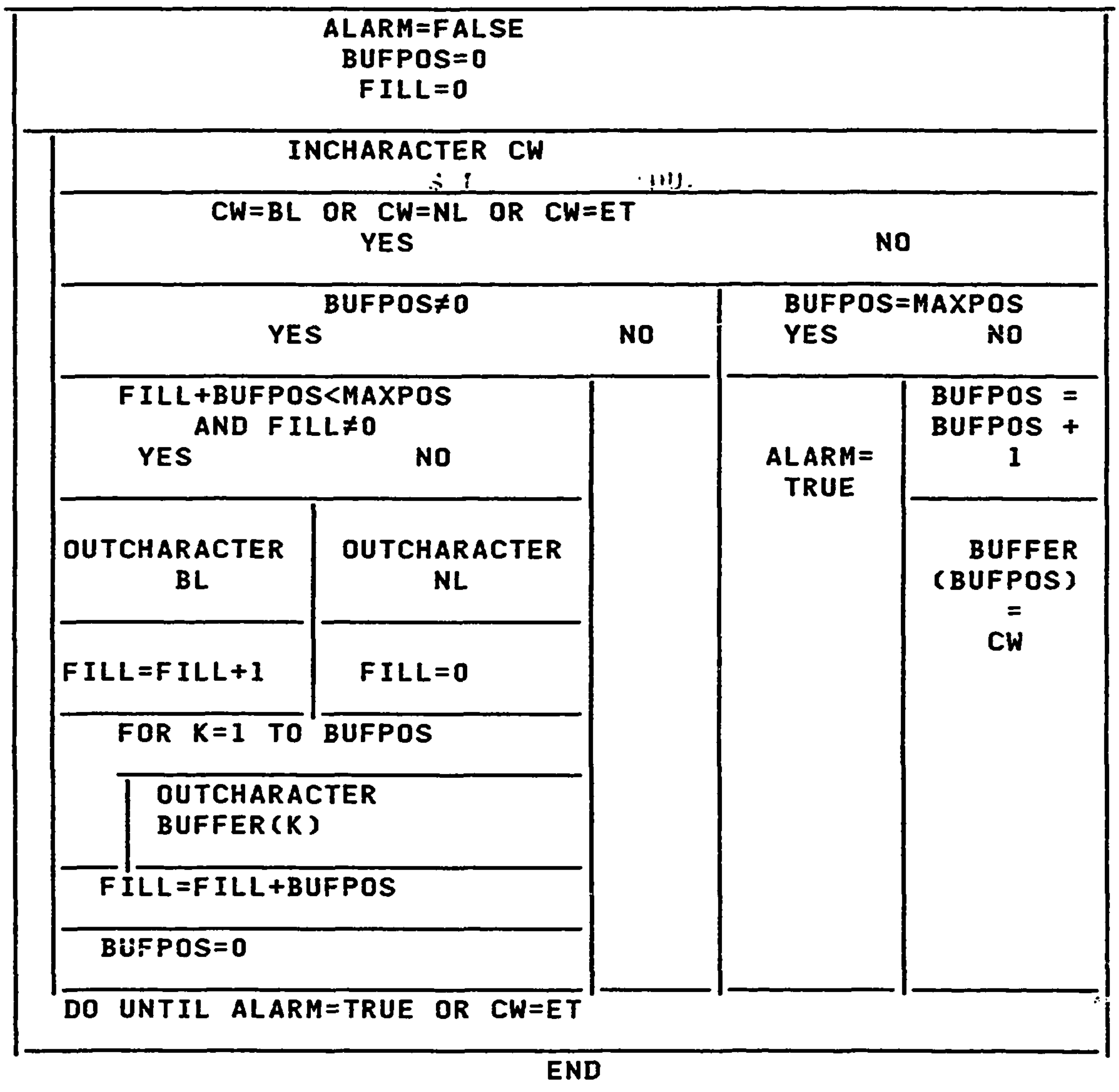
ACTUAL VERSUS PREDICTED FOR INDIVIDUAL C'S

OBS	ACTUAL	PREDICT VALUE	STD ERR PREDICT	RESIDUAL	STD ERR RESIDUAL	STUDENT RESIDUAL
1	0.400000	0.480102	0.046896	-.080102	0.118067	-0.678
2	0.600000	0.543850	0.041259	0.056150	0.120153	0.467
3	0.600000	0.590934	0.039238	0.009066	0.120828	0.075
4	0.600000	0.866236	0.039555	-.266236	0.120725	-2.205
5	0.800000	0.864469	0.040852	-.064469	0.120292	-0.536
6	0.800000	0.906936	0.040763	-.106936	0.120322	-0.889
7	0.710000	0.758859	0.036977	-.048859	0.121539	-0.402
8	0.860000	0.861504	0.035019	-.001504	0.122118	-0.012
9	0.860000	0.906936	0.040763	-.046936	0.120322	-0.390
10	0.280000	0.294630	0.103110	-.014630	0.074211	-0.197
11	0.860000	0.907313	0.038428	-.047313	0.121088	-0.391
12	0.670000	0.558624	0.046264	0.111376	0.118316	0.941
13	1.000	0.909881	0.039394	0.090119	0.120777	0.746
14	1.000	0.906936	0.040763	0.093064	0.120322	0.773
15	0.830000	0.768931	0.034362	0.061069	0.122304	0.499
16	0.570000	0.401958	0.051074	0.168042	0.116321	1.445
17	0.710000	0.620899	0.037183	0.089101	0.121476	0.733
18	0.850000	0.828131	0.064095	0.021869	0.109685	0.199
19	1.000	0.816350	0.068865	0.183650	0.106755	1.720
20	0.710000	0.711485	0.051959	-.001485	0.115928	-0.013
21	0.500000	0.508113	0.039004	-.008113	0.120904	-0.067
22	0.500000	0.599615	0.038917	-.099615	0.120932	-0.824
23	1.000	0.795037	0.063912	0.204963	0.109792	1.867
24	0.500000	0.567408	0.106181	-.067408	0.069747	-0.966
25	0.330000	0.400393	0.050746	-.070393	0.116464	-0.604
26	0.330000	0.515543	0.047557	-.185543	0.117802	-1.575
27	0.660000	0.834641	0.065815	-.174641	0.108662	-1.607

```
PLOT OF RESIDUALS FOR INDIVIDUAL C PREDICTOR

    2.0 +
        I
        I
    1.5 +
        I
        I
    1.0 +
R       I
E       I
S   0.5 +
I       I
D       I           *            *    *            ***      **
U   0.0 +---*-------*-----*--*--**-*---------*---**---*--**-**--
A       I                    *      *                  *     *
L       I                                                *
S  -0.5 +
        I
        I
   -1.0 +
        I
        I
   -1.5 +
        I
        I
   -2.0 +
        I
        I
   -2.5 +
        I
        I
   -3.0 +-------------------------------------------------------
        I
        I
        I
        I
        I
        ----+-------+-------+-------+-------+-------+-------+---
           0.3     0.4     0.5     0.6     0.7     0.8     0.9

                          PREDICTED VALUE
```



```
ALARM=FALSE
BUFPOS=0
FILL=0
    INCHARACTER CW
    CW=BL OR CW=NL OR CW=ET
    YES
        BUFPOS≠0
        YES
            FILL+BUFPOS<MAXPOS
            AND FILL≠0
            YES
                OUTCHARACTER
                BL
                FILL=FILL+1
            NO
                OUTCHARACTER
                NL
                FILL=0
            FOR K=1 TO BUFPOS
                OUTCHARACTER
                BUFFER(K)
            FILL=FILL+BUFPOS
            BUFPOS=0
        NO
    NO
        BUFPOS=MAXPOS
        YES
            ALARM=
            TRUE
        NO
            BUFPOS =
            BUFPOS +
            1
            BUFFER
            (BUFPOS)
            =
            CW
DO UNTIL ALARM=TRUE OR CW=ET
END
```

BIBLIOGRAPHY

Acree, A. T. Jr.; On Mutation, Ph.D. Dissertation, Georgia Institute of Technology, 1980.

Adrion, W. R., Branstrad, M. A., Cherniavsky J. C; Validation, Verification, and Testing of Computer Software, Computing Surveys, ACM, Vol. 14, No. 2, June 1982.

Alberts, D. S.; The Economics of Software Quality Assurance, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Boehm, B. W., McClean, R. K. and Urfrig D. B.; Some Experience with Automated Aids to the Design of Large-Scale Reliable Software, Tutorial: Software Testing and Validation Techniques, IEEE,1978.

Boehm, B. W., The High Cost of Software, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Budd, T. A., Mutation Analysis of Program Test Data, Ph.D Dissertation, Yale University, May 1980.

Chapin, N., New Format for Flowcharts, Software - Practice and Experience, Vol. 4, 1974.

Chudleigh, M.; Software Reliability, Systems International, July 1982.

Clarke, L. A.; A System to Generate Test Data and Symbolically Execute Programs, IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976.

Clarke, L. A., Hassell, J., Richardson D. J.; A Close Look at Domain Testing, IEEE Transactions on Software Engineering, Vol. SE-6, No. 4, July 1982.

DeMillo, R. A.; Mutation Analysis as a Tool for Software Quality Assurance, Proceedings of COMPSAC 80, IEEE, 1980.

DeMillo, R. A., Program Mutation: An Approach to Software Testing, Report AD-A135775, Georgia Institute Of Technology, April 1983.

Deutsch, L. P.; An Interactive Program Verifier, Ph.D. Dissertation, University of California, Berkeley, 1973.

Duran, J. W., Wiorkowski, J. J.; Toward Models for Probabilistic Program Correctness, Proceeding of the Software Quality and Assurance Workshop, ACM, November 1978.

Elmendorf, W. R.; Cause-Effect Graphs In Functional Testing, IBM Technical Report TR 00.2487, November 1973. ACM Computing Surveys, Vol. 8, No. 3, September, 1976.

Fagan, M. E.; Design and Code Inspections to Reduce Errors in Program Development, Tutorial: Software Testing and Validation Techniques, IEEE, 1978, also published as, Design and Code Inspections and Process Control in the Development of Programs, IBM Technical Report TR00.2763, June 1976.

Fosdick, L. D., Osterweil, L. J.; Data Flow Analysis in Software Reliability, Computing Surveys, Vol. 8, No. 3, September 1976.

Friedman, A. D., Menon, P. R.; Fault Detection in Digital Circuits, Englewood, N.J., Prentice Hall, 1979.

Gabow, H. N.; Maheshwari, Osterweil, L. J.; On Two Problems in the Generation of Program Test Paths, IEEE Transactions On Software Engineering, SE-2, No. 3 September 1976.

Gillion, P., Why are Users Getting Untested Programs?, Computerworld ,Vol. XVII, No. 31, August 1983.

Glass, R. J.; Software Reliability Guidebook, Prentice Hall 1979.

Goodenough, J. B. and Gerhart, S. L.; Toward a Theory of Test Data Selection, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, also in, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Goodenough, J. B.; A Survey of Program Testing Issues, Chapter 9, Research Directions in Computer Science, The MIT Press, 1980.

Hansen, P. B.; Testing a Multiprogramming System, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Hiedler, W., Benson, J., Meeson, J., Kerbel, A.; Pyster, A.; Software Testing Measures, Rome Air Development Center, Report RADC-TR-82-135, May 1982.

Hogger, E. I., A Decision Table Approach to Reliable Software, IAEA/Westinghouse Conference on Software Reliability, 1977.

Howden, W. E.; Symbolic Testing and the DISSECT Symbolic Evaluation System, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Howden W. E.; A Survey of Static Analysis Methods, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Howden, W. E.; Reliability of The Path Analysis Testing Strategy, IEEE Transactions on Software Engineering, Vol.SE-2, No. 3, September 1976, also in, Tutorial: Software Testing and Validation Techniques, IEEE, 1978.

Howden, W. E.; Weak Mutation Testing and Completeness of Test Sets, IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, July 1982.

Huang, J. C.; An Approach to Program Testing, _Computing Surveys,_ Vol. 7, No. 3, September 1975, also in, _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978.

Kernighan, B. W., Plauger, P. J.; _The Elements of Programning Style,_ McGraw - Hill, 1974.

Manna, Z., Waldinger, R.; The Logic of Computer Programming, _IEEE Transactions on Software Engineering,_ Vol. SE-4, No. 3 May 1978.

McCracken, D. D., _A Simplified Guide to Fortran Programming_ ,John Wiley, 1974.

Miller, E. F. Jr; Program Testing: Art Meets Theory, _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978, also in _Computer,_ July 1977.

Miller, E. F. Jr.; Program Testing Technology in the 1980's, _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978

Miller E. F. Jr., Melton R. A.; Automated Generation of Testcase Datasets, _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978.

Miller E. F. Jr., Paige M. R., Benson J. P., and Wisehart W. R.; Structural Techniques of Program Validation _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978.

Mills, H. B., On the Statistical Validation of Computer Programs, FSC-72-6015, IBM, 1972.

Musa, J. D.; The Measurement and Management of Software Reliability, _Proceeding of the IEEE,_ Vol. 68, No. 9, September 1980.

Myers, G. J.; _Software Reliability, Principles and Practices,_ Wiley - Interscience, 1976.

Neumann, P. G.; Some Computer-Related Disasters and Other Egregious Horrors, _ACM SIGSOFT Software Engineering Notes,_ Vol. 10, No. 1, January 1985.

Osterweil, L. J.; A Proposal for an Integrated Testing System for Computer Programs, University of Colorado, Technical Report, October 1976.

Osterweil L. J., Fosdick, L. D.; DAVE-A Validation Error Detection and Documentation System for Fortran Programs, _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978.

Phoha, S.; A Quantifiable Methodology for Software Testing: Using Path Analysis, Project 4130, The MITRE Corp., December 1981.

Probert, R. L.; Optimal Insertion of Software Probes in Well- Delimited Programs, _IEEE Transactions on Software Engineering,_ Vol. SE-8, No. 1, January 1982.

Ramamoorthy, C. V., Ho, S. F.; Testing Large Software with Automated Software Evaluation Systems, _Tutorial: Software Testing and Validation Techniques,_ IEEE, 1978.

Ramamoorthy, C. V.; Ho, S.F.; Chen, W. T.; On the Automated Generation of Program Test Data, _IEEE Transactions on Software Engineering,_ Vol. SE-2, No.4 December 1979.

Ramamoorthy, C. V., Bastani, F.; Software Reliability - Status and Perspectives, _IEEE Transactions on Software Engineering,_ Vol. SE-2, No. 4, July 1982.

Reynolds, J. C.; Proving Program Correctness, Rome Air Development Center Report, RADC-TR-08-379, Vol. V, 1980.

SAS Institute Inc. _SAS User's Guide: Statistics_ , 1982.

Shankar, K. S.; A Functional Approach to Model Verification, _IEEE Transactions of Software Engineering,_ Vol. SE-8, No. 2, March 1982.

Stickney, M. E.; An application of Graph Theory to Software Test Data Selection, _Proceeding of the Software_

Quality and Assurance Workshop, ACM, November 1978.

Tai, K.; Program Testing Complexity and Test Criteria, Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980.

Walsh P. J.;An Analysis of Test Case Selection, Phoenix Conference Proceedings, IEEE, March 1983.

Wang; J. S.; Measuring Completeness of a Test Case Library, IBM Technical Disclosure Bulletin, Vol. 23, No. 9 February 1981.

White, L. H., Cohen E. I.; A Domain Strategy for Computer Programming Testing, IEEE Transactions on Software Engineering, SE-6, May 1980.

Wilson, P. B., Building Quality into Software with Effective Testing, Small Systems World, August, 1983.

Yaacob, M. and Hartley M. G.; A Survey Of Microprocessor Reliability with an Illustrative Example, Int. J. Elec. Eng. Education, Vol. 18, Pg 159-174, 1981.